

AD-A252 405

FORMATION PAGE

Form Approved  
OPM No.Public  
and n  
sugg  
2220.

age 1 hour per response, including the time for reviewing instructions, searching existing data sources gathering  
1. Send comments regarding this burden estimate or any other aspect of this collection of information, including  
Directorate for Information Operations and Reports, 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA  
if Management and Budget, Washington, DC 20503.

1. AGENCY USE (Leave)		2. REPORT		3. REPORT TYPE AND DATES Final: 11 May 1992 to 01 Jun 1993	
4. TITLE AND Validation Summary Report:Alenia Aeritalia & Selenia S.p.A, DACS VAX/VMS to 80x86 PM MARA Ada Cross Compiler, Version 4.6, MicroVAX 4000/200 (Host) to MARA (Target), 920503S1.11259				5. FUNDING <div style="text-align: center;">2</div>	
6. National Institute of Standards and Technology Gaithersburg, MD USA					
7. PERFORMING ORGANIZATION NAME(S) AND National Institute of Standards and Technology National Computer Systems Laboratory Bldg. 255, Rm A266 Gaithersburg, MD 20899 USA				8. PERFORMING ORGANIZATION NIST92ALE505_1_1.11	
9. SPONSORING/MONITORING AGENCY NAME(S) AND Ada Joint Program Office United States Department of Defense Pentagon, RM 3E114 Washington, D.C. 20301-3081				10. SPONSORING/MONITORING AGENCY	
11. SUPPLEMENTARY <div style="text-align: center;"><b>DTIC</b> <b>S ELECTE D</b> <b>JUL 01 1992</b> <b>A</b></div>					
12a. DISTRIBUTION/AVAILABILITY Approved for public release; distribution unlimited.				12b. DISTRIBUTION	
13. (Maximum 200 Alenia Aeritalia & Selenia S.p.A, DACS VAX/VMS to 80x86 PM MARA Ada Cross Compiler, Version 4.6, MicroVAX 4000/200 (Host) to MARA (Target), ACVC 1.11. <div style="text-align: right; font-size: 24pt; font-weight: bold;">92-17186</div> <div style="text-align: center;"></div> <div style="text-align: center; font-size: 24pt; font-weight: bold;">92 0 003</div>					
14. SUBJECT Ada programming language, Ada Compiler Val. Summary Report, Ada Compiler Val. Capability, Val. Testing, Ada Val. Office, Ada Val. Facility, ANS/MIL-STD-1815A,				15. NUMBER OF	
				16. PRICE	
17. SECURITY CLASSIFICATION UNCLASSIFIED		18. SECURITY UNCLASSIFIED		19. SECURITY CLASSIFICATION UNCLASSIFIED	
20. LIMITATION OF					

NSN

Standard Form 298, (Rev. 2-89)  
Prescribed by ANSI Std.

AVF Control Number: NIST92ALE505\_1\_1.11  
DATE COMPLETED  
BEFORE ON-SITE: 92-05-02  
AFTER ON-SITE: 92-05-11  
REVISIONS:

Ada COMPILER  
VALIDATION SUMMARY REPORT:  
Certificate Number: 920509S1.11259  
Alenia Aeritalia & Selenia S.p.A  
DACS VAX/VMS to 80x86 PM MARA Ada Cross Compiler, Version 4.6  
MicroVAX 4000/200 => MARA

Prepared By:  
Software Standards Validation Group  
National Computer Systems Laboratory  
National Institute of Standards and Technology  
Building 225, Room A266  
Gaithersburg, Maryland 20899

Accession For	
NTIS CRA&I	<input checked="checked" type="checkbox"/>
DTIC TAB	<input type="checkbox"/>
Unannounced	<input type="checkbox"/>
Justification	
By	
Distribution /	
Availability Codes	
Dist	Avail and/or Special
A-1	



AVF Control Number: NIST92ALE505\_1\_1.11

Certificate Information

The following Ada implementation was tested and determined to pass ACVC 1.11. Testing was completed on May 09, 1992.

Compiler Name and Version: DACS VAX/VMS to 80x86 PM MARA Ada Cross Compiler, Version 4.6

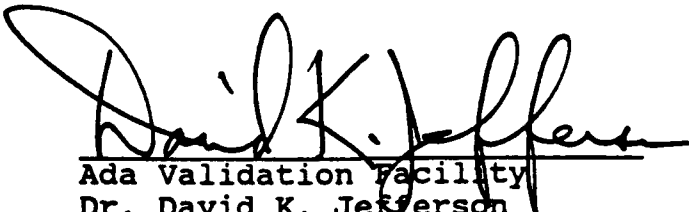
Host Computer System: MicroVAX 4000/200 running VAX/VMS, Version 5.4

Target Computer System: MARA (Alenia computer based on INTEL 80286 running Alenia Operating System, Version 8.6 System)

See section 3.1 for any additional information about the testing environment.

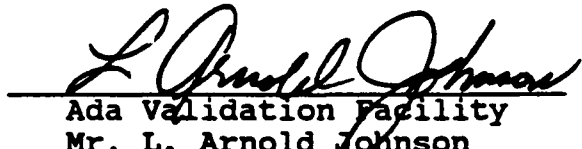
As a result of this validation effort, Validation Certificate 920509S1.11259 is awarded to Alenia Aeritalia & Selenia S.p.A. This certificate expires on [the re-RE-revised Common Expiration Date: 2 years post ANSI/MIL-STD-1815B standardization].

This report has been reviewed and is approved.

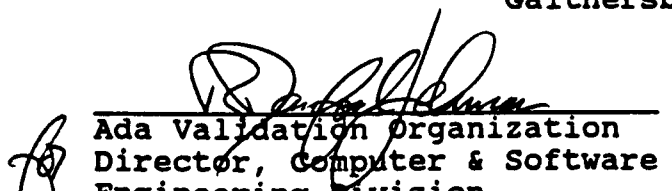


Ada Validation Facility  
Dr. David K. Jefferson  
Chief, Information Systems  
Engineering Division (ISED)

Computer Systems Laboratory (CLS)  
National Institute of Standards and Technology  
Building 225, Room A266  
Gaithersburg, MD 20899



Ada Validation Facility  
Mr. L. Arnold Johnson  
Manager, Software Standards  
Validation Group



Ada Validation Organization  
Director, Computer & Software  
Engineering Division  
Institute for Defense Analyses  
Alexandria VA 22311



Ada Joint Program Office  
Dr. John Solomond  
Director  
Department of Defense  
Washington DC 20301

DECLARATION OF CONFORMANCE

The following declaration of conformance was supplied by the customer.

Customer: Alenia Aeritalia & Selenia S.p.A

Certificate Awardee: Alenia Aeritalia & Selenia S.p.A

Ada Validation Facility: National Institute of Standards and  
Technology  
Computer Systems Laboratory (CSL)  
Software Validation Group  
Building 225, Room A266  
Gaithersburg, Maryland 20899

ACVC Version: 1.11

Ada Implementation:

Compiler Name and Version: DACS VAX/VMS to 80x86 PM MARA Ada  
Cross Compiler, Version 4.6

Host Computer System: MACROVAX 4000/200 running VAX/VMS,  
Version 5.4

Target Computer System: MARA (Alenia computer based on INTEL  
80286 running Alenia Operating  
System, Version 8.6 System)

Declaration:

I, the undersigned, declare that I have no knowledge of deliberate  
deviations from the Ada Language Standard ANSI/MIL-STD-1815A ISO  
8652-1987 in the implementation listed above.

  
\_\_\_\_\_  
Customer Signature

Company Alenia Aeritalia & Selenia S.p.A  
Title: Director

8-5-92  
Date

  
\_\_\_\_\_  
Certificate Awardee Signature

Company Alenia Aeritalia & Selenia S.p.A  
Title: Director

8-5-92  
Date

## TABLE OF CONTENTS

CHAPTER 1 . . . . .	1-1
INTRODUCTION . . . . .	1-1
1.1 USE OF THIS VALIDATION SUMMARY REPORT . . . . .	1-1
1.2 REFERENCES . . . . .	1-1
1.3 ACVC TEST CLASSES . . . . .	1-2
1.4 DEFINITION OF TERMS . . . . .	1-3
CHAPTER 2 . . . . .	2-1
IMPLEMENTATION DEPENDENCIES . . . . .	2-1
2.1 WITHDRAWN TESTS . . . . .	2-1
2.2 INAPPLICABLE TESTS . . . . .	2-1
2.3 TEST MODIFICATIONS . . . . .	2-4
CHAPTER 3 . . . . .	3-1
PROCESSING INFORMATION . . . . .	3-1
3.1 TESTING ENVIRONMENT . . . . .	3-1
3.2 SUMMARY OF TEST RESULTS . . . . .	3-1
3.3 TEST EXECUTION . . . . .	3-2
APPENDIX A . . . . .	A-1
MACRO PARAMETERS . . . . .	A-1
APPENDIX B . . . . .	B-1
COMPILATION SYSTEM OPTIONS . . . . .	B-1
LINKER OPTIONS . . . . .	B-2
APPENDIX C . . . . .	C-1
APPENDIX F OF THE Ada STANDARD . . . . .	C-1

## CHAPTER 1

### INTRODUCTION

The Ada implementation described above was tested according to the Ada Validation Procedures [Pro90] against the Ada Standard [Ada83] using the current Ada Compiler Validation Capability (ACVC). This Validation Summary Report (VSR) gives an account of the testing of this Ada implementation. For any technical terms used in this report, the reader is referred to [Pro90]. A detailed description of the ACVC may be found in the current ACVC User's Guide [UG89].

#### 1.1 USE OF THIS VALIDATION SUMMARY REPORT

Consistent with the national laws of the originating country, the Ada Certification Body may make full and free public disclosure of this report. In the United States, this is provided in accordance with the "Freedom of Information Act" (5 U.S.C. #552). The results of this validation apply only to the computers, operating systems, and compiler versions identified in this report.

The organizations represented on the signature page of this report do not represent or warrant that all statements set forth in this report are accurate and complete, or that the subject implementation has no nonconformities to the Ada Standard other than those presented. Copies of this report are available to the public from the AVF which performed this validation or from:

National Technical Information Service  
5285 Port Royal Road  
Springfield VA 22161

Questions regarding this report or the validation test results should be directed to the AVF which performed this validation or to:

Ada Validation Organization  
Computer and Software Engineering Division  
Institute for Defense Analyses  
1801 North Beauregard Street  
Alexandria VA 22311-1772

#### 1.2 REFERENCES

[Ada83] Reference Manual for the Ada Programming Language,  
ANSI/MIL-STD-1815A, February 1983 and ISO 8652-1987.

[Pro90] Ada Compiler Validation Procedures, Version 2.1, Ada Joint Program Office, August 1990.

[UG89] Ada Compiler Validation Capability User's Guide, 21 June 1989.

### 1.3 ACVC TEST CLASSES

Compliance of Ada implementations is tested by means of the ACVC. The ACVC contains a collection of test programs structured into six test classes: A, B, C, D, E, and L. The first letter of a test name identifies the class to which it belongs. Class A, C, D, and E tests are executable. Class B and class L tests are expected to produce errors at compile time and link time, respectively.

The executable tests are written in a self-checking manner and produce a PASSED, FAILED, or NOT APPLICABLE message indicating the result when they are executed. Three Ada library units, the packages REPORT and SPRT13, and the procedure CHECK\_FILE are used for this purpose. The package REPORT also provides a set of identity functions used to defeat some compiler optimizations allowed by the Ada Standard that would circumvent a test objective. The package SPRT13 is used by many tests for Chapter 13 of the Ada Standard. The procedure CHECK\_FILE is used to check the contents of text files written by some of the Class C tests for Chapter 14 of the Ada Standard. The operation of REPORT and CHECK\_FILE is checked by a set of executable tests. If these units are not operating correctly, validation testing is discontinued.

Class B tests check that a compiler detects illegal language usage. Class B tests are not executable. Each test in this class is compiled and the resulting compilation listing is examined to verify that all violations of the Ada Standard are detected. Some of the class B tests contain legal Ada code which must not be flagged illegal by the compiler. This behavior is also verified.

Class L tests check that an Ada implementation correctly detects violation of the Ada Standard involving multiple, separately compiled units. Errors are expected at link time, and execution is attempted.

In some tests of the ACVC, certain macro strings have to be replaced by implementation-specific values -- for example, the largest integer. A list of the values used for this implementation is provided in Appendix A. In addition to these anticipated test modifications, additional changes may be required to remove unforeseen conflicts between the tests and implementation-dependent characteristics. The modifications required for this implementation are described in section 2.3.

For each Ada implementation, a customized test suite is produced by the AVF. This customization consists of making the modifications described in the preceding paragraph, removing withdrawn tests (see section 2.1) and, possibly some inapplicable tests (see Section 3.2 and [UG89]).

In order to pass an ACVC an Ada implementation must process each test of the customized test suite according to the Ada Standard.

#### 1.4 DEFINITION OF TERMS

Ada Compiler	The software and any needed hardware that have to be added to a given host and target computer system to allow transformation of Ada programs into executable form and execution thereof.
Ada Compiler Validation Capability (ACVC)	The means for testing compliance of Ada implementations, Validation consisting of the test suite, the support programs, the ACVC Capability user's guide and the template for the validation summary (ACVC) report.
Ada Implementation	An Ada compiler with its host computer system and its target computer system.
Ada Joint Program (AJPO)	The part of the certification body which provides policy and guidance for the Ada certification Office system.
Ada Validation Facility (AVF)	The part of the certification body which carries out the procedures required to establish the compliance of an Ada implementation.
Ada Validation Organization (AVO)	The part of the certification body that provides technical guidance for operations of the Ada certification system.
Compliance of an Ada Implementation	The ability of the implementation to pass an ACVC version.
Computer System	A functional unit, consisting of one or more computers and associated software, that uses common storage for all or part of a program and also for all or part of the data necessary for the execution of the program; executes user-written or user-designated programs; performs user-designated data manipulation, including



arithmetic operations and logic operations; and that can execute programs that modify themselves during execution. A computer system may be a stand-alone unit or may consist of several inter-connected units.

Conformity	Fulfillment by a product, process or service of all requirements specified.
Customer	An individual or corporate entity who enters into an agreement with an AVF which specifies the terms and conditions for AVF services (of any kind) to be performed.
Declaration of Conformance	A formal statement from a customer assuring that conformity is realized or attainable on the Ada implementation for which validation status is realized.
Host Computer System	A computer system where Ada source programs are transformed into executable form.
Inapplicable test	A test that contains one or more test objectives found to be irrelevant for the given Ada implementation.
ISO	International Organization for Standardization.
LRM	The Ada standard, or Language Reference Manual, published as ANSI/MIL-STD-1815A-1983 and ISO 8652-1987. Citations from the LRM take the form "<section>.<subsection>:<paragraph>."
Operating System	Software that controls the execution of programs and that provides services such as resource allocation, scheduling, input/output control, and data management. Usually, operating systems are predominantly software, but partial or complete hardware implementations are possible.
Target Computer System	A computer system where the executable form of Ada programs are executed.
Validated Ada Compiler	The compiler of a validated Ada implementation.
Validated Ada Implementation	An Ada implementation that has been validated successfully either by AVF testing or by registration [Pro90].

Validation

The process of checking the conformity of an Ada compiler to the Ada programming language and of issuing a certificate for this implementation.

Withdrawn  
test

A test found to be incorrect and not used in conformity testing. A test may be incorrect because it has an invalid test objective, fails to meet its test objective, or contains erroneous or illegal use of the Ada programming language.

## CHAPTER 2

### IMPLEMENTATION DEPENDENCIES

#### 2.1 WITHDRAWN TESTS

Some tests are withdrawn by the AVO from the ACVC because they do not conform to the Ada Standard. The following 95 tests had been withdrawn by the Ada Validation Organization (AVO) at the time of validation testing. The rationale for withdrawing each test is available from either the AVO or the AVF. The publication date for this list of withdrawn tests is 91-08-02.

E28005C	B28006C	C32203A	C34006D	C35508I	C35508J
C35508M	C35508N	C35702A	C35702B	B41308B	C43004A
C45114A	C45346A	C45612A	C45612B	C45612C	C45651A
C46022A	B49008A	B49008B	A74006A	C74308A	B83022B
B83022H	B83025B	B83025D	B83026B	C83026A	C83041A
B85001L	C86001F	C94021A	C97116A	C98003B	BA2011A
CB7001A	CB7001B	CB7004A	CC1223A	BC1226A	CC1226B
BC3009B	BD1B02B	BD1B06A	AD1B08A	BD2A02A	CD2A21E
CD2A23E	CD2A32A	CD2A41A	CD2A41E	CD2A87A	CD2B15C
BD3006A	BD4008A	CD4022A	CD4022D	CD4024B	CD4024C
CD4024D	CD4031A	CD4051D	CD5111A	CD7004C	ED7005D
CD7005E	AD7006A	CD7006E	AD7201A	AD7201E	CD7204B
AD7206A	BD8002A	BD8004C	CD9005A	CD9005B	CDA201E
CE2107I	CE2117A	CE2117B	CE2119B	CE2205B	CE2405A
CE3111C	CE3116A	CE3118A	CE3411B	CE3412B	CE3607B
CE3607C	CE3607D	CE3812A	CE3814A	CE3902B	

#### 2.2 INAPPLICABLE TESTS

A test is inapplicable if it contains test objectives which are irrelevant for a given Ada implementation. The inapplicability criteria for some tests are explained in documents issued by ISO and the AJPO known as Ada Commentaries and commonly referenced in the format AI-ddddd. For this implementation, the following tests were determined to be inapplicable for the reasons indicated; references to Ada Commentaries are included as appropriate.

The following 201 tests have floating-point type declarations requiring more digits than SYSTEM.MAX\_DIGITS:

C24113L..Y (14 tests)	C35705L..Y (14 tests)
C35706L..Y (14 tests)	C35707L..Y (14 tests)
C35708L..Y (14 tests)	C35802L..Z (15 tests)

C45241L..Y (14 tests)	C45321L..Y (14 tests)
C45421L..Y (14 tests)	C45521L..Z (15 tests)
C45524L..Z (15 tests)	C45621L..Z (15 tests)
C45641L..Y (14 tests)	C46012L..Z (15 tests)

C24113I..K (3 TESTS) use a line length in the input file which exceeds 126 characters.

C35404D, C45231D, B86001X, C86006E, and CD7101G check for a predefined integer type with a name other than INTEGER, LONG\_INTEGER, or SHORT\_INTEGER; for this implementation, there is no such type.

C35713B, C45423B, B86001T, and C86006H check for the predefined type SHORT\_FLOAT; for this implementation, there is no such type.

C35713D and B86001Z check for a predefined floating-point type with a name other than FLOAT, LONG\_FLOAT, or SHORT\_FLOAT; for this implementation, there is no such type.

C45531M..P and C45532M..P (8 tests) check fixed-point operations for types that require a SYSTEM.MAX\_MANTISSA of 47 or greater; for this implementation, MAX\_MANTISSA is less than 47.

C45624A..B (2 tests) check that the proper exception is raised if MACHINE\_OVERFLOW is FALSE for floating point types and the results of various floating-point operations lie outside the range of the base type; for this implementation, MACHINE\_OVERFLOW is TRUE.

C4A013B contains a static universal real expression that exceeds the range of this implementation's largest floating-point type; this expression is rejected by the compiler.

D56001B uses 65 levels of block nesting; this level of block nesting exceeds the capacity of the compiler.

C96005B uses values of type DURATION's base type that are outside the range of type DURATION; for this implementation, the ranges are the same.

CA2009C and CA2009F check whether a generic unit can be instantiated before its body (and any of its subunits) is compiled; this implementation creates a dependence on generic units as allowed by AI-00408 and AI-00506 such that the compilation of the generic unit bodies makes the instantiating units obsolete. (See section 2.3.)

CD1009C checks whether a length clause can specify a non-default size for a floating-point type; this implementation does not support such sizes.

CD2A84A, CD2A84E, CD2A84I..J (2 tests), and CD2A84O use length clauses to specify non-default sizes for access types; this implementation does not support such sizes.

The 19 tests listed in the following table are not applicable because the given file operations are supported for the given combination of mode and file access method.

Test	File Operation	Mode	File Access Method
CE2102E	CREATE	OUT_FILE	SEQUENTIAL_IO
CE2102F	CREATE	INOUT_FILE	DIRECT_IO
CE2102J	CREATE	OUT_FILE	DIRECT_IO
CE2102N	OPEN	IN_FILE	SEQUENTIAL_IO
CE2102O	RESET	IN_FILE	SEQUENTIAL_IO
CE2102P	OPEN	OUT_FILE	SEQUENTIAL_IO
CE2102Q	RESET	OUT_FILE	SEQUENTIAL_IO
CE2102R	OPEN	INOUT_FILE	DIRECT_IO
CE2102S	RESET	INOUT_FILE	DIRECT_IO
CE2102T	OPEN	IN_FILE	DIRECT_IO
CE2102U	RESET	IN_FILE	DIRECT_IO
CE2102V	OPEN	OUT_FILE	DIRECT_IO
CE2102W	RESET	OUT_FILE	DIRECT_IO
CE3102F	RESET	Any Mode	TEXT_IO
CE3102G	DELETE	-----	TEXT_IO
CE3102I	CREATE	OUT_FILE	TEXT_IO
CE3102J	OPEN	IN_FILE	TEXT_IO
CE3102K	OPEN	OUT_FILE	TEXT_IO
CE3109A	CREATE	IN_FILE	TEXT_IO

The 2 tests listed in the following table check the given file operations for the given combination of mode and access method; this implementation does not support these operations.

Test	File Operation	Mode	File Access Method
CE2105A	CREATE	IN_FILE	SEQUENTIAL_IO
CE2105B	CREATE	IN_FILE	DIRECT_IO

The following 15 tests check operations on sequential, direct, and text files when multiple internal files are associated with the same external file; USE\_ERROR is raised when this association is attempted.

CE2107A..B	CE2107E..G	CD2110B	CE2110D	CE2111D
CE2111H	CE3111A..B	CE3111D..E	CE3114B	CE3115A

CE2107C..D (2 tests), CE2107H, and CE2107L apply function NAME to temporary sequential, direct, and text files in an attempt to associate multiple internal files with the same external file; USE\_ERROR is raised because temporary files have no name.

CE2108B, CE2108D, and CE3112B use the names of temporary sequential, direct, and text files that were created in other tests in order to check that the temporary files are not accessible after the completion of those tests; for this implementation, temporary files have no name.

CE2203A checks that WRITE raises USE\_ERROR if the capacity of an external sequential file is exceeded; this implementation cannot restrict file capacity.

EE2401D uses an instantiation of DIRECT\_IO with an unconstrained array type; for this implementation, the maximum element size of the array type exceeds the implementation limit of 32Kbytes and so USE\_ERROR is raised.

CE2403A checks that WRITE raises USE\_ERROR if the capacity of an external direct file is exceeded; this implementation cannot restrict file capacity.

CE3304A checks that SET\_LINE\_LENGTH and SET\_PAGE\_LENGTH raise USE\_ERROR if they specify an inappropriate value for the external file; there are no inappropriate values for this implementation.

CE3413B checks that PAGE raises LAYOUT\_ERROR when the value of the page number exceeds COUNT'LAST; for this implementation, the value of COUNT'LAST is greater than 150000, making the checking of this objective impractical.

## 2.3 TEST MODIFICATIONS

Modifications (see section 1.3) were required for 68 tests.

The following tests were split into two or more tests because this implementation did not report the violations of the Ada Standard in the way expected by the original tests.

B22003A	B26001A	B26002A	B26005A	B28003A	B29001A	B33301B
B35101A	B37106A	B37301B	B37302A	B38003A	B38003B	B38009A
B38009B	B55A01A	B61001C	B61001F	B61001H	B61001I	B61001M
B61001R	B61001W	B67001H	B83A07A	B83A07B	B83A07C	B83E01C
B83E01D	B83E01E	B85001D	B85008D	B91001A	B91002A	B91002B
B91002C	B91002D	B91002E	B91002F	B91002G	B91002H	B91002I
B91002J	B91002K	B91002L	B95030A	B95061A	B95061F	B95061G
B95077A	B97103E	B97104G	BA1001A	BA1101B	BC1109A	BC1109C
BC1109D	BC1202A	BC1202F	BC1202G	BE2210A	BE2413A	

C83030C and C86007A were graded passed by Test Modification as directed by the AVO. These tests were modified by inserting "PRAGMA ELABORATE (REPORT);" before the package declarations at lines 13 and 11, respectively. Without the pragma, the packages may be elaborated prior to package Report's body, and thus the packages' calls to function REPORT.IDENT\_INT at lines 14 and 13, respectively, will raise PROGRAM\_ERROR.

CA2009C and CA2009F were graded inapplicable by Evaluation Modification as directed by the AVO. These tests contain instantiations of a generic unit prior to the compilation of that unit's body; as allowed by AI-00408 and AI-00506, the compilation of the generic unit bodies makes the compilation unit that contains the instantiations obsolete.

BC3204C and BC3205D were graded passed by Processing Modification as directed by the AVO. These tests check that instantiations of generic units with unconstrained types as generic actual parameters are illegal if the generic bodies contain uses of the types that require a constraint. However, the generic bodies are compiled after the units that contain the instantiations, and this implementation creates a dependence of the instantiating units on the generic units as allowed by AI-00408 and AI-00506 such that the compilation of the generic bodies makes the instantiating units obsolete--no errors are detected. The processing of these tests was modified by re-compiling the obsolete units; all intended errors were then detected by the compiler.

## CHAPTER 3

### PROCESSING INFORMATION

#### 3.1 TESTING ENVIRONMENT

The Ada implementation tested in this validation effort is described adequately by the information given in the initial pages of this report.

For technical information about this Ada implementation, contact:

Dr. Nicola Botta  
Alenia Aeritalia & Selenia S.p.A  
Via Tiburtina km. 12,4  
00131 Roma, Italy  
Telephone ++39 6 41972520  
Telex 613690 / 616180 Alroma I  
Fax ++39 6 4131452

For sales information about this Ada implementation, contact:

Dr. Renato Ciabattoni  
Alenia Aeritalia & Selenia S.p.A  
Via Tiburtina km. 12,4  
00131 Roma, Italy  
Telephone ++39 6 41973277  
Telex 613690 / 616180 Alroma I  
Fax ++39 6 4131452

Testing of this Ada implementation was conducted at the customer's site by a validation team from the AVF.

#### 3.2 SUMMARY OF TEST RESULTS

An Ada Implementation passes a given ACVC version if it processes each test of the customized test suite in accordance with the Ada Programming Language Standard, whether the test is applicable or inapplicable; otherwise, the Ada Implementation fails the ACVC [Pro90].

For all processed tests (inapplicable and applicable), a result was obtained that conforms to the Ada Programming Language Standard.

The list of items below gives the number of ACVC tests in various categories. All tests were processed, except those that were withdrawn because of test errors (item b; see section 2.1). All



tests passed, except those that are listed in sections 2.1 and 2.2 (counted in items b and f, below).

a) Total Number of Applicable Tests	3792	
b) Total Number of Withdrawn Tests	95	
c) Processed Inapplicable Tests	283	
d) Non-Processed I/O Tests	0	
e) Non-Processed Floating-Point Precision Tests	0	
f) Total Number of Inapplicable Tests	283	(c+d+e)
g) Total Number of Tests for ACVC 1.11	4170	(a+b+f)

### 3.3 TEST EXECUTION

A magnetic tape containing the customized test suite (see section 1.3) was taken on-site by the validation team for processing. The contents of the magnetic tape were loaded directly onto the host computer.

After the test files were loaded onto the host computer, the full set of tests was processed by the Ada implementation.

Communications between the VAX, host computer system, and the MARA, target computer system, is done via Ethernet link using TCP/IP protocols, and two Alenia proprietary communications software programs: TOP and MJC. TOP is used to initialize the target. MJC is used to load the executable module(s) and to capture the execution results.

The tests were compiled and linked on the host computer system, as appropriate. The executable images were transferred to the target computer system by the communications link described above, and run. The results were captured on the host computer system.

Testing was performed using command scripts provided by the customer and reviewed by the validation team. See Appendix B for a complete listing of the processing options for this implementation. It also indicates the default options. The options invoked explicitly for validation testing during this test were:

For B tests, E tests, CZ, and not\_applicable tests:  
/LIST /NOSAVE\_SOURCE

For all other tests:  
/NOSAVE\_SOURCE

Test output, compiler and linker listings, and job logs were captured on magnetic tape and archived at the AVF. The listings examined on-site by the validation team were also archived.

# APPENDIX A

## MACRO PARAMETERS

This appendix contains the macro parameters used for customizing the ACVC. The meaning and purpose of these parameters are explained in [UG89]. The parameter values are presented in two tables. The first table lists the values that are defined in terms of the maximum input-line length, which is the value for \$MAX\_IN\_LEN--also listed here. These values are expressed here as Ada string aggregates, where "V" represents the maximum input-line length.

Macro Parameter	Macro Value
\$MAX_IN_LEN	126 -- Value of V
\$BIG_ID1	(1..V-1 => 'A', V => '1')
\$BIG_ID2	(1..V-1 => 'A', V => '2')
\$BIG_ID3	(1..V/2 => 'A') & '3' & (1..V-1-V/2 => 'A')
\$BIG_ID4	(1..V/2 => 'A') & '4' & (1..V-1-V/2 => 'A')
\$BIG_INT_LIT	(1..V-3 => '0') & "298"
\$BIG_REAL_LIT	(1..V-5 => '0') & "690.0"
\$BIG_STRING1	'"' & (1..V/2 => 'A') & '"'
\$BIG_STRING2	'"' & (1..V-1-V/2 => 'A') & '1' & '"'
\$BLANKS	(1..V-20 => ' ')
\$MAX_LEN_INT_BASED_LITERAL	"2:" & (1..V-5 => '0') & "11:"
\$MAX_LEN_REAL_BASED_LITERAL	"16:" & (1..V-7 => '0') & "F.E:"
\$MAX_STRING_LITERAL	'"' & (1..V-2 => 'A') & '"'

The following table contains the values for the remaining macro parameters.

Macro Parameter	Macro Value
ACC_SIZE	: 32
ALIGNMENT	: 1
COUNT_LAST	: 2_147_483_647
DEFAULT_MEM_SIZE	: 1_048_576
DEFAULT_STOR_UNIT	: 16
DEFAULT_SYS_NAME	: IAPX286
DELTA_DOC	: 2#1.0#E-31
ENTRY_ADDRESS	: (140,0)
ENTRY_ADDRESS1	: (141,0)
ENTRY_ADDRESS2	: (142,0)
FIELD_LAST	: 67
FILE_TERMINATOR	: ASCII.CR & ASCII.LF & ASCII.FF
FIXED_NAME	: NO_SUCH_FIXED_TYPE
FLOAT_NAME	: SHORT_SHORT_FLOAT
FORM_STRING	: ""
FORM_STRING2	:
	"CANNOT RESTRICT_FILE_CAPACITY"
GREATER_THAN_DURATION	: 75_000.0
GREATER_THAN_DURATION_BASE_LAST	: 131_073.0
GREATER_THAN_FLOAT_BASE_LAST	: 16#1.0#E+32
GREATER_THAN_FLOAT_SAFE_LARGE	: 16#5.FFFF_F0#E+31
GREATER_THAN_SHORT_FLOAT_SAFE_LARGE	: 1.0E308
HIGH_PRIORITY	: 7
ILLEGAL_EXTERNAL_FILE_NAME1	: ILL-FILE
ILLEGAL_EXTERNAL_FILE_NAME2	:
	THISFILENAMEISTOOLONGFORMYSYSTEM
INAPPROPRIATE_LINE_LENGTH	: -1
INAPPROPRIATE_PAGE_LENGTH	: -1
INCLUDE_PRAGMA1	:
	PRAGMA INCLUDE ("A28006D1.TST")
INCLUDE_PRAGMA2	:
	PRAGMA INCLUDE ("B28006E1.TST")
INTEGER_FIRST	: -32768
INTEGER_LAST	: 32767
INTEGER_LAST_PLUS_1	: 32768
INTERFACE_LANGUAGE	: ASM86
LESS_THAN_DURATION	: -75_000.0
LESS_THAN_DURATION_BASE_FIRST	: -131_073.0
LINE_TERMINATOR	: ASCII.CR & ASCII.LF
LOW_PRIORITY	: 0
MACHINE_CODE_STATEMENT	:
	MACHINE_INSTRUCTION' (NONE, m_RETN);
MACHINE_CODE_TYPE	: REGISTER_TYPE

MANTISSA_DOC	: 31
MAX_DIGITS	: 15
MAX_INT	: 2147483647
MAX_INT_PLUS_1	: 2147483648
MIN_INT	: -2147483648
NAME	: NO_SUCH_TYPE_AVAILABLE
NAME_LIST	: IAPX286
NAME_SPECIFICATION1	: :FMS:MARALAB/X2120A
NAME_SPECIFICATION2	: :FMS:MARALAB/X2120B
NAME_SPECIFICATION3	: :FMS:MARALAB/X3119A
NEG_BASED_INT	: 16#FFFFFFFF#
NEW_MEM_SIZE	: 1_048_576
NEW_STOR_UNIT	: 16
NEW_SYS_NAME	: IAPX286
PAGE_TERMINATOR	: ASCII.FF
RECORD_DEFINITION	: RECORD NULL;END RECORD;
RECORD_NAME	: NO_SUCH_MACHINE_CODE_TYPE
TASK_SIZE	: 16
TASK_STORAGE_SIZE	: 1024
TICK	: 0.000_000_125
VARIABLE_ADDRESS	: (16#0#,16#3C#)
VARIABLE_ADDRESS1	: (16#4#,16#3C#)
VARIABLE_ADDRESS2	: (16#8#, 16#3C#)
YOUR_PRAGMA	: EXPORT_OBJECT

## APPENDIX B

### COMPILATION SYSTEM OPTIONS

The compiler options of this Ada implementation, as described in this Appendix, are provided by the customer. Unless specifically noted otherwise, references in this appendix are to compiler documentation and not to this report.

QUALIFIER	DESCRIPTION	REFERENCES
<code>/AUTO_INLINE</code> <code>/NOAUTO_INLINE</code>	Specifies whether local subprograms should be inline expanded.	5.1.1
<code>/CHECK</code> <code>/NOCHECK</code>	Controls run-time checks.	5.1.2
<code>/CONFIGURATION_FILE</code>	Specifies the configuration file used by the compiler.	5.1.3
<code>/DEBUG</code> <code>/NODEBUG</code>	Includes symbolic debugging information in program Library. Does not include symbolic information.	5.1.4
<code>/EXCEPTION_TABLES</code> <code>/NOEXCEPTION_TABLES</code>	Includes/excludes exception handler tables from the generated code.	5.1.15
<code>/FIXPOINT_ROUNDING</code> <code>/NOFIXPOINT_ROUNDING</code>	Generates fixed point rounding code. Avoids fixed point rounding code.	5.1.6
<code>/FLOAT_ALLOWED</code> <code>/NOFLOAT_ALLOWED</code>	Flags generation of float instructions as error if selected.	5.1.7
<code>/LIBRARY</code>	Specifies program library used.	5.1.8
<code>/LIST</code> <code>/NOLIST</code>	Writes a source listing on the list file.	5.1.9
<code>/OPTIMIZE</code> <code>/NOOPTIMIZE</code>	Specifies compiler optimization.	5.1.10
<code>/PROGRESS</code> <code>/NOPROGRESS</code>	Displays compiler progress.	5.1.11

<b>/SAVE_SOURCE</b>	Copies source to program library.	5.1.12
<b>/NO\$AVE_SOURCE</b>		
<b>/TARGET_DEBUG</b>	Includes Intel debug information.	5.1.5
<b>/NOTARGET_DEBUG</b>	Does not include Intel debug information.	
<b>/XREF</b>	Creates a cross reference listing.	5.1.13
<b>/NOXREF</b>		
<b>/UNIT</b>	Assigns a specific unit number to the compilation (must be free and in a sublibrary).	5.1.14

---

## LINKER OPTIONS

The linker options of this Ada implementation, as described in this Appendix, are provided by the customer. Unless specifically noted otherwise, references in this appendix are to linker documentation and not to this report.

### Linker Configuration Qualifiers

QUALIFIER	DESCRIPTION	REFERENCE
<b>/DEBUG</b> <b>/NODEBUG</b>	Links an application for use with the Cross Debugger.	6.5.6
<b>/LIBRARY</b>	The library used in the link.	6.5.2
<b>/LOG</b> <b>/NOLOG</b>	Specifies creation of a log file.	6.5.4
<b>/OPTIONS</b>	Specifies target link options.	6.5.1
<b>/SEARCHLIB</b>	Target libraries or object modules to include in target link	6.1.4
<b>/ROOT_EXTRACT</b> <b>/NOROOT_EXTRACT</b>	Using non-DDC-I units in the root library	6.5.5
<b>/SELECTIVE_LINK</b>	Removes uncalled code from final program.	6.5.3
<b>/STOP_BEFORE_LINK</b>	Performs Ada prelink only.	6.1.5
<b>/OFD= &lt;object file directory&gt;</b>	Default may be set to the logical name "OFD:". The name of the directory to contain object files.	14.3
<b>/AUTO_CLUSTER</b>	Auto clusterization is active, if this qualifier is set.	14.3
<b>/CLUSTER= &lt;cluster file&gt;</b>	Takes cluster information from file specified.	14.3
<b>/OPTIMIZE</b>	Assures that intra-cluster CALLs are optimized. Far CALLs are substituted for PUSH CS, near CALL sequences. Returns remain far.	14.3



<b>/PROCESSOR_ASSIGN= &lt;proc.assign_file&gt;</b>	Takes information about cluster position on processors into account, when elaborating.	14.3
<b>/INTERFACED= &lt;file&gt;</b>	The file or library specified will be added to the link command. An alternative to this qualifier is to use /SEARCHLIB (see above).	14.3
<b>/RELINK= &lt;relink_file&gt;</b>	The file is supposed to contain lines describing units that have been recompiled (under the same unit numbers).	14.3
<b>/PRELINK</b>	Activates Ada prelink.	APFX, Part II, 1.1.1
<b>/LINK</b>	Enables to obtain an executable program (.LTL) through a unique command in which appropriate qualifiers specify (see [User's Guide]) the parameters to be passed to the various tools involved in the generation.	APFX, Part II, 1.1.1
<b>/TEMPLATE [ = &lt;identifier&gt; ]</b>	Allows to specify which program architecture in the <graph> file specified by 'ada'_graph, is required to generate an executable program.	APFX, Part II, 1.1.2.1
<b>/MAP /NOMAP (default)</b>	It is necessary to create the map(<main>.MGA) of the program to be generated.	APFX, Part II, 1.1.2.2

#### Run-Time System Configuration Qualifiers

QUALIFIER	DESCRIPTION	REFERENCE
<b>/LT_STACK_SIZE</b>	Library task default stack size	7.2.4
<b>/LT_SEGMENT_SIZE</b>	Library task default segment size	7.2.5
<b>/MP_STACK_SIZE</b>	Main program stack size	7.2.6
<b>/MP_SEGMENT_SIZE</b>	Main program segment size	7.2.7
<b>/PRIORITY</b>	Default task priority	7.2.1
<b>/TASK_STORAGE_SIZE</b>	Tasks default storage size	7.2.8

## APPENDIX C

### APPENDIX F OF THE Ada STANDARD

The only allowed implementation dependencies correspond to implementation-dependent pragmas, to certain machine-dependent conventions as mentioned in Chapter 13 of the Ada Standard, and to certain allowed restrictions on representation clauses. The implementation-dependent characteristics of this Ada implementation, as described in this Appendix, are provided by the customer. Unless specifically noted otherwise, references in this Appendix are to compiler documentation and not to this report. Implementation-specific portions of the package STANDARD, which are not a part of Appendix F, are:

package STANDARD is

type SHORT\_INTEGER is range -128 .. 127;

type INTEGER is range -32\_768 .. 32\_767;

type LONG\_INTEGER is range -2\_147\_483\_648 .. 2\_147\_483\_647;

type FLOAT is digits 6

range -16#0.FFFF\_FF#E32 .. 16#0.FFFF\_FF#E32;

type LONG\_FLOAT is digits 15

range -16#0.FFFF\_FFFF\_FFFF\_F8#E256 .. 16#0.FFFF\_FFFF\_FFFF\_F8#E256;

type DURATION is delta 2#1.0#E-14 range -131\_072.0 .. 131\_071.0;

end STANDARD;

## APPENDIX F IMPLEMENTATION-DEPENDENT CHARACTERISTICS

This appendix describes the implementation-dependent characteristics of DACS-80X86™ as required in Appendix F of the Ada Reference Manual (ANSI/MIL-STD-1815A).

### A. Implementation-Dependent Pragmas

This section describes all implementation defined pragmas.

#### 1. Pragma `INTERFACE_SPELLING`

This pragma allows an Ada program to call a non-Ada program whose name contains characters that are invalid in Ada subprogram identifiers. This pragma must be used in conjunction with pragma `INTERFACE`, i.e., pragma `INTERFACE` must be specified for the Ada subprogram name prior to using pragma `INTERFACE_SPELLING`.

The pragma has the format:

```
pragma INTERFACE_SPELLING (subprogram name, string literal);
```

where the subprogram name is that of one previously given in pragma `INTERFACE` and the string literal is the exact spelling of the interfaced subprogram in its native language. This pragma is only required when the subprogram name contains invalid characters for Ada identifiers.

Example:

```
function RTS_GetDataSegment return Integer;  
  
pragma INTERFACE (ASM86, RTS_GetDataSegment);  
pragma INTERFACE_SPELLING (RTS_GetDataSegment,  
                             "R1SMGS?GetDataSegment");
```

The string literal may be appended 'NEAR (or 'FAR) to specify a particular method of call. The default is 'FAR. This suffix should only be used, when the called routines require a near call (writing 'FAR is however harmless). If 'NEAR is added, the routine must be in the same segment as the caller.

User's Guide  
Implementation-Dependent Characteristics

## 2. Pragma LT\_SEGMENT\_SIZE

This pragma sets the size of a library task stack segment.  
The pragma has the format:

```
pragma LT_SEGMENT_SIZE (T, N);
```

where T denotes either a task object or task type and N designates the size of the library task stack segment in words.

The library task's stack segment defaults to the size of the library task stack. The size of the library task stack is normally specified via the representation clause (note that T must be a task type)

```
for T'SORAGE_SIZE use N;
```

The size of the library task stack segment determines how many tasks can be created which are nested within the library task. All tasks created within a library task will have their stacks allocated from the same segment as the library task stack. Thus, pragma LT\_SEGMENT\_SIZE must be specified to reserve space within the library task stack segment so that nested tasks' stacks may be allocated (see section ?).

The following restrictions are places on the use of LT\_SEGMENT\_SIZE:

1. It must be used only for library tasks.
2. It must be placed immediately after the task object or type name declaration.
3. The library task stack segment size (N) must be greater than or equal to the library task stack size.

## 3. Pragma EXTERNAL\_NAME

### a. Function

The pragma EXTERNAL\_NAME is designed to make permanent Ada objects and subprograms externally available using names supplied by the user.

## User's Guide Implementation-Dependent Characteristics

### b. Format

The format of the pragma is:

```
pragma EXTERNAL_NAME(<ada_entity>,<external name>)
```

where <ada\_entity> should be the name of:

- a permanent object, i.e. an object placed in the permanent pool of the compilation unit - such objects originate from package specifications and bodies only,
- a constant object, i.e. an object placed in the constant pool of the compilation unit - please note that scalar constants are embedded in the code, and composite constants are not always placed in the constant pool, because the constant is not considered constant by the compiler,
- a subprogram name, i.e. a name of a subprogram defined in this compilation unit - please notice that separate subprogram specifications cannot be used, the code for the subprogram must be present in the compilation unit code, and where the <external name> is a string specifying the external name associated the <ada\_entity>. The <external names> should be unique. Specifying identical spellings for different <ada\_entities> will generate errors at compile and/or link time, and the responsibility for this is left to the user. Also the user should avoid spellings similar to the spellings generated by the compiler, e.g. E\_xxxxx\_yyyyy, P\_xxxxx, C\_xxxxx and other internal identifications. The target debug type information associated with such external names is the null type.

### c. Restrictions

Objects that are local variables to subprograms or blocks cannot have external names associated. The entity being made external ("public") must be defined in the compilation unit itself. Attempts to name entities from other compilation units will be rejected with a warning.

When an entity is an object the value associated with the symbol will be the relocatable address of the first byte assigned to the object.

### d. Example

Consider the following package body fragment:

```
package body example is
    subtype string10 is string(1..10);
    type s is
        record
            len : integer;
            val : string10;
        end record;
```

## User's Guide

### Implementation-Dependent Characteristics

```
global_s : s;  
const_s  : constant string10 := "1234567890";  
  
pragma EXTERNAL_NAME(global_s, "GLOBAL_S_OBJECT");  
pragma EXTERNAL_NAME(const_s, "CONST_S");  
  
procedure handle(...) is  
...  
end handle;  
  
pragma EXTERNAL_NAME(handle, "HANDLE_PROC");  
  
...  
  
end example;
```

The objects GLOBAL\_S and CONST\_S will have associated the names "GLOBAL\_S\_OBJECT" and "CONST\_S". The procedure HANDLE is now also known as "HANDLE\_PROC". It is allowable to assign more than one external name to an Ada entity.

#### e. Object Layouts

Scalar objects are laid out as described in Chapter 9. For arrays the object is described by the address of the first element; the array constraint(s) are NOT passed, and therefore it is recommended only to use arrays with known constraints. Non-discriminated records take a consecutive number of bytes, whereas discriminated records may contain pointers to the heap. Such complex objects should be made externally visible, only if the user has thorough knowledge about the layout.

#### f. Parameter Passing

The following section describes briefly the fundamentals regarding parameter passing in connection with Ada subprograms. For more detail, refer to Chapter 9.

Scalar objects are always passed by value. For OUT or IN OUT scalars, code is generated to move the modified scalar to its destination. In this case the stack space for parameters is not removed by the procedure itself, but by the caller.

Composite objects are passed by reference. Records are passed via the address of the first byte of the record. Constrained arrays are passed via the address of the first byte (plus a bitoffset when a packed array). Unconstrained arrays are passed as constrained arrays plus a pointer to the constraints for each index in the array. These constraints consist of lower and upper bounds, plus the size in words or bits of each element depending if the value is positive or negative respectively. The user should study an appropriate disassembler listing to thoroughly understand the compiler calling conventions.

A function (which can only have IN parameters) returns its result in register(s). Scalar results are registers/float registers only; composite results leave an address in some registers and the rest, if any, are placed on the stack top. The stack still contains the parameters in this case (since the function result is likely to be on the stack), so the caller must restore the stack pointer to a suitable value, when the function call is dealt with. Again, disassemblies may guide the user to see how a particular function call is to be handled.

## User's Guide

### Implementation-Dependent Characteristics

#### B. Implementation-Dependent Attributes

No implementation-dependent attributes are defined.

#### C. Package SYSTEM

The specifications of package SYSTEM for all DACS-80x86 in Real Address Mode and DACS-80286PM systems are identical.

Below is package system for DACS-80x86.

package System is

```
type Word is new Integer;
type DWord is new Long_integer;
```

```
type UnsignedWord is range 0..65535;
for UnsignedWord'SIZE use 16;
```

```
type byte is range 0..255;
for byte'SIZE use 8;
```

```
subtype SegmentId is UnsignedWord;
```

```
type Address is
  record
    offset : UnsignedWord;
    segment : SegmentId;
  end record;
```

```
subtype Priority is Integer range 0..7;
type Name is (IAPX286);
```

```
SYSTEM_NAME : constant Name := IAPX286;
STORAGE_UNIT : constant := 16;
MEMORY_SIZE : constant := 1 048 576;
MIN_INT : constant := -2 147 483 647-1;
MAX_INT : constant := 2 147 483 647;
MAX_DIGITS : constant := 15;
MAX_MANTISSA : constant := 31;
FINE_DELTA : constant := 2#1.0#E-31;
TICK : constant := 0.000_000_125;
```

## User's Guide

### Implementation-Dependent Characteristics

```

type Interface_language is
    (ASM86,      PLM86,      C86,      C86 REVERSE,
     ASM_ACF,    PLM_ACF,    C_ACF,    C REVERSE ACF,
     ASM_NOACF,  PLM_NOACF,  C_NOACF,  C REVERSE_NOACF);

type ExceptionId is record
    unit_number : UnsignedWord;
    unique_number : UnsignedWord;
end record;

type TaskValue is new Integer;
type AccTaskValue is access TaskValue;
type SemaphoreValue is new Integer;

type Semaphore is record
    counter : Integer;
    first : TaskValue;
    last : TaskValue;
end record;

InitSemaphore : constant Semaphore := Semaphore'(1,0,0);
foreign_exception : exception;
end System;

```

#### D. Representation Clauses

The DACS-80x86™ fully supports the 'SIZE representation for derived types. The representation clauses that are accepted for non-derived types are described in the following subsections.

##### 1. Length Clause

Some remarks on implementation dependent behavior of length clauses are necessary:

- When using the SIZE attribute for discrete types, the maximum value that can be specified is 16 bits. For DACS-80386PM/80486PM the maximum is 32 bits.
- SIZE is only obeyed for discrete types when the type is a part of a composite object, e.g. arrays or records, for example:

```

type byte is range 0..255;
for byte'size use 8;

```

```

sixteen_bits_allocated : byte;                                -- one word allocated

```

```

eight_bit_per_element : array(0..7) of byte;                -- four words allocated

```

```

type rec is

```

```

    record

```

```

        c1,c2 : byte;

```

```

    end record;

```

-- eight bits per component



**User's Guide**  
**Implementation-Dependent Characteristics**

- Using the **STORAGE\_SIZE** attribute for a collection will set an upper limit on the total size of objects allocated in this collection. If further allocation is attempted, the exception **STORAGE\_ERROR** is raised.
- When **STORAGE\_SIZE** is specified in a length clause for a task type, the process stack area will be of the specified size. The process stack area will be allocated inside the "standard" stack segment. Note that **STORAGE\_SIZE** may not be specified for a task object.

## **2. Enumeration Representation Clauses**

Enumeration representation clauses may specify representations in the range of -32767..+32766 (or -16#7FFF..16#7FFE).

## **3. Record Representation Clauses**

When representation clauses are applied to records the following restrictions are imposed:

- if the component is a **record** or an **unpacked array**, it must start on a storage unit boundary (16 bits)
- a record occupies an integral number of storage units (words) (even though a record may have fields that only define an odd number of bytes)
- a record may take up a maximum of 32K bits
- a component must be specified with its proper size (in bits), regardless of whether the component is an array or not (Please note that record and unpacked array components take up a number of bits divisible by 16 (=word size))
- if a non-array component has a size which equals or exceeds one storage unit (16 bits) the component must start on a storage unit boundary, i.e. the component must be specified as:

component     at N range 0..16 \* M - 1;

where N specifies the relative storage unit number (0,1,...) from the beginning of the record, and M the required number of storage units (1,2,...)

- the elements in an array component should always be wholly contained in one storage unit
- if a component has a size which is less than one storage unit, it must be wholly contained within a single storage unit:

component     at N range X .. Y;

## User's Guide

### Implementation-Dependent Characteristics

where  $N$  is as in previous paragraph, and  $0 \leq X \leq Y \leq 15$ . Note that for this restriction a component is not required to start in an integral number of storage units from the beginning of the record.

If the record type contains components which are not covered by a component clause, they are allocated consecutively after the component with the value. Allocation of a record component without a component clause is always aligned on a storage unit boundary. Holes created because of component clauses are not otherwise utilized by the compiler.

Pragma pack on a record type will attempt to pack the components not already covered by a representation clause (perhaps none). This packing will begin with the small scalar components and larger components will follow in the order specified in the record. The packing begins at the first storage unit after the components with representation clauses.

#### a. Alignment Clauses

Alignment clauses for records are implemented with the following characteristics:

- If the declaration of the record type is done at the outermost level in a library package, any alignment is accepted.
- If the record declaration is done at a given static level higher than the outermost library level, i.e., the permanent area), only word alignments are accepted.
- Any record object declared at the outermost level in a library package will be aligned according to the alignment clause specified for the type. Record objects declared elsewhere can only be aligned on a word boundary. If the record type is associated with a different alignment, an error message will be issued.
- If a record type with an associated alignment clause is used in a composite type, the alignment is required to be one word; an error message is issued if this is not the case.

#### E. Implementation-Dependent Names for Implementation Dependent Components

None defined by the compiler.

#### F. Address Clauses

This section describes the implementation of address clauses and what types of entities may have their address specified by the user.

**User's Guide**  
**Implementation-Dependent Characteristics**

**1. Objects**

Address clauses are supported for scalar and composite objects whose size can be determined at compile time. The address clause may denote a dynamic value.

**G. Unchecked Conversion**

Unchecked conversion is only allowed between objects of the same "size". However, if scalar type has different sizes (packed and unpacked), unchecked conversion between such a type and another type is accepted if either the packed or the unpacked size fits the other type.

**User's Guide**  
**Implementation-Dependent Characteristics**

## **H. Machine Code Insertions**

The reader should be familiar with the code generation strategy and the 80x86 instruction set to fully benefit from this section.

As described in chapter 13.8 of the ARM [DoD 83] it is possible to write procedures containing only code statements using the predefined package `MACHINE_CODE`. The package `MACHINE_CODE` defines the type `MACHINE_INSTRUCTION` which, used as a record aggregate, defines a machine code insertion. The following sections list the type `MACHINE_INSTRUCTION` and types on which it depends, give the restrictions, and show an example of how to use the package `MACHINE_CODE`.

### **1. Predefined Types for Machine Code Insertions**

The following types are defined for use when making machine code insertions (their type declarations are given on the following pages):

```
type opcode_type
type operand_type
type register_type
type segment_register
```

## User's Guide

### Implementation-Dependent Characteristics

type machine\_instruction

The type REGISTER\_TYPE defines registers. The registers STi describe registers on the floating stack. (ST is the top of the floating stack).

The type MACHINE\_INSTRUCTION is a discriminant record type with which every kind of instruction can be described. Symbolic names may be used in the form

name'ADDRESS

Restrictions as to symbolic names can be found in section F.8.2.

It should be mentioned that addresses are specified as 80386/80486 addresses. In case of other targets, the scale factor should be set to "scale\_1".

```
type opcode_type is (
  -- 8086 instructions:
  m_AAA, m_AAD, m_AAM, m_AAS, m_ADC, m_ADD, m_AND, m_CALL, m_CALLN,
  m_CBW, m_CLC, m_CLD, m_CLI, m_CMC, m_CMP, m_CMPS, m_CWD, m_DAA,
  m_DAS, m_DEC, m_DIV, m_HLT, m_IDIV, m_IMUL, m_IN, m_INC, m_INT,
  m_INT0, m_IRET, m_JA, m_JAE, m_JB, m_JBE, m_JC, m_JCXZ, m_JE,
  m_JG, m_JGE, m_JL, m_JLE, m_JNA, m_JNAE, m_JNB, m_JNBE, m_JNC,
  m_JNE, m_JNG, m_JNL, m_JNLE, m_JNO, m_JNP, m_JNS, m_JNZ,
  m_JO, m_JP, m_JPE, m_JPO, m_JS, m_JZ, m_JMP, m_LAHF, m_LDS,
  m_LES, m_LEA, m_LOCK, m_LODS, m_LOOP, m_LOOPE, m_LOOPNE, m_LOOPNZ,
  m_LOOPZ, m_MOV, m_MOVS, m_MUL, m_NEG, m_NOP, m_NOT, m_OR, m_OUT,
  m_POP, m_POPF, m_PUSH, m_PUSHD, m_RCL, m_RCR, m_ROL, m_ROR,
  m_REP, m_REPE, m_REPN, m_RET, m_RETF, m_RETN, m_RETF, m_SAHF,
  m_SAL, m_SAR, m_SHL, m_SHR, m_SBB, m_SCAS, m_STC, m_STD, m_STI,
  m_STOS, m_SUB, m_TEST, m_WAIT, m_XCHG, m_XLAT, m_XOR,

  -- 8087/80187/80287 Floating Point Processor instructions:
  m_FABS, m_FADD, m_FADDD, m_FADDP, m_FBLD, m_FBSTP, m_FCHS,
  m_FNCLEX, m_FCOM, m_FCOMD, m_FCOMP, m_FCOMP, m_FCOMPP, m_FDECSTP,
  m_FDIV, m_FDIVD, m_FDIVP, m_FDIVR, m_FDIVRD, m_FDIVRP, m_FFREE,
  m_FIADD, m_FIADDD, m_FICOM, m_FICOMD, m_FICOMP, m_FICOMP, m_FIDIV,
  m_FIDIVD, m_FIDIVR, m_FIDIVRD, m_FILD, m_FILDD, m_FILD, m_FIMUL,
  m_FIMULD, m_FINCSTP, m_FINIT, m_FIST, m_FISTD, m_FISTP, m_FISTPD,
  m_FISTPL, m_FISUB, m_FISUBD, m_FISUBR, m_FISUBRD, m_FLD, m_FLDD,
  m_FLD, m_FLDENV, m_FLDLG2, m_FLDLN2, m_FLDL2E, m_FLDL2T, m_FLDPI,
  m_FLDZ, m_FLD1, m_FML, m_FMLD, m_FMLP, m_FNOP, m_FPTAN,
  m_FPREM, m_FPTAN, m_FRNDINT, m_FRSTOR, m_FSAVE, m_FSCALE, m_FSETPM,
  m_FSQRT, m_FST, m_FSTD, m_FSTCW, m_FSTENV, m_FSTP, m_FSTPD,
  m_FSTSW, m_FSTSWAX, m_FSUB, m_FSUBD, m_FSUBP, m_FSUBR, m_FSUBRD,
  m_FSUBRP, m_FTST, m_FWAIT, m_FXAM, m_FXCH, m_FTRACT, m_FYL2X,
  m_FYL2XP1, m_F2XM1,

  -- 80186/80286/80386 instructions:
  -- Notice that some immediate versions of the 8086
  -- instructions only exist on these targets
  -- (shifts, rotates, push, imul, ...)
  m_BOUND, m_CLTS, m_ENTER, m_INS, m_IAR, m_LEAVE, m_LGDT,
  m_LIDT, m_LSL, m_OUTS, m_POPA, m_PUSHA, m_SGDT, m_SIDT,
  m_ARPL, m_LLDT, m_LMSW, m_LTR,

  -- 16 bit always...
  m_SLDT, m_SMSW, m_STR, m_VERR, m_VERN,

  -- the 80386 specific instructions:
  m_SETA, m_SETAE, m_SETB, m_SETBE, m_SETC, m_SETE,
  m_SETG, m_SETGE, m_SETL, m_SETLE, m_SETNA, m_SETNAE,
  m_SETNB, m_SETNBE, m_SETNC, m_SETNE, m_SETNG,
  m_SETNGE, m_SETNL, m_SETNLE, m_SETNO, m_SETNP, m_SETNS,
```

## User's Guide

### Implementation-Dependent Characteristics

```

m_SETNZ, m_SETO, m_SETP, m_SETPE, m_SETPO, m_SETS,
m_SETZ, m_BSF, m_BSR, m_BT, m_BTC, m_BTR,
m_BTS, m_LFS, m_LGS, m_LSS, m_MOVZX, m_MOVSX,
m_MOVCX, m_MOVDB, m_MOVTR, m_SHLD, m_SHRD,

```

-- the 80387 specific instructions:

```

m_FUCOM, m_FUCOMP, m_FUCOMPP, m_FPREM1, m_FSIN, m_FCOS,
m_FSINCOS,

```

-- byte/w ord/dword variants (to be used, when  
-- not deductible from context):

```

m_ADCB, m_ADCW, m_ADCD, m_ADDB, m_ADDW, m_ADDD,
m_ANDB, m_ANDW, m_ANDD, m_BIT, m_BTD, m_BTCW,
m_BTCD, m_BTRW, m_BTRD, m_BTSW, m_BTSD, m_CBW,
m_CWDE, m_CWDW, m_CDQ, m_CMPB, m_CMPW, m_CMPD,
m_CMPSB, m_CMPSW, m_CMPSD, m_DECB, m_DECW, m_DECD,
m_DIVB, m_DIVW, m_DIVD, m_IDIB, m_IDIW, m_IDID,
m_IMULB, m_IMULW, m_IMULD, m_INCB, m_INCW, m_INCD,
m_INSB, m_INSW, m_INSD, m_LODSB, m_LODSW, m_LODSD,
m_MOVB, m_MOVW, m_MOVD, m_MOVB, m_MOVS, m_MOVSD,
m_MOVSXB, m_MOVSXW, m_MOVZX, m_MOVZXW, m_MULB, m_MULW,
m_MULD, m_NEGB, m_NEGW, m_NEGD, m_NOTB, m_NOTW,
m_NOTD, m_ORB, m_ORW, m_ORD, m_OUTSB, m_OUTSW,
m_OUTSD, m_POPW, m_POPD, m_PUSHW, m_PUSD, m_RCLB,
m_RCLW, m_RCLD, m_RCRB, m_RCRW, m_RCRD, m_ROLB,
m_ROLW, m_ROLD, m_RORB, m_RORW, m_RORD, m_SALB,
m_SALW, m_SALD, m_SARB, m_SARW, m_SARD, m_SHLB,
m_SHLW, m_SHLDW, m_SHRB, m_SHRW, m_SHRDW, m_SBBB,
m_SBBW, m_SBB, m_SCASB, m_SCASW, m_SCASD, m_STOSB,
m_STOSW, m_STOSD, m_SUBB, m_SUBW, m_SUBD, m_TESTB,
m_TESTW, m_TESTD, m_XORB, m_XORW, m_XORD, m_DATAB,
m_DATAW, m_DATAD,

```

-- Special 'instructions': m\_label, m\_reset,

-- 8087 temp real load/store\_and\_pop: m\_FLD, m\_FSTPT;

pragma page;

type operand\_type is ( none, -- no operands

immediate, -- one immediate operand

register, -- one register operand

address, -- one address operand

system\_address, -- one 'address operand

name, -- CALL name

register\_immediate, -- two operands :

-- destination is

-- register

-- source is immediate

register\_register, -- two register operands

register\_address, -- two operands :

-- destination is

-- register

-- source is address

address\_register, -- two operands :

-- destination is

-- address

-- source is register

register\_system\_address, -- two operands :

-- destination is

-- register

-- source is 'address

system\_address\_register, -- two operands :

-- destination is

-- 'address

-- source is register

address\_immediate, -- two operands :

-- destination is

-- address

-- source is immediate

system\_address\_immediate, -- two operands :

-- destination is

## User's Guide

### Implementation-Dependent Characteristics

```

-- 'address
-- source is immediate
immediate_register, -- only allowed for OUT
-- port is immediate
-- source is register
immediate_immediate, -- only allowed for
-- ENTER
register_register_immediate, -- allowed for IMULimm,
-- SHRDimm, SHLDimm
register_address_immediate, -- allowed for IMULimm
register_system_address_immediate, -- allowed for IMULimm
address_register_immediate, -- allowed for SHRDimm,
-- SHLDimm
system_address_register_immediate -- allowed for SHRDimm,
-- SHLDimm
);

type register_type is
(AX, CX, DX, BX, SP, BP, SI, DI, -- word regs
AL, CL, DL, BL, AH, CH, DH, BH, -- byte regs
EAX, ECX, EDX, EBX, ESP, EBP, ESI, EDI, -- dword regs
ES, CS, SS, DS, FS, GS, -- selectors
BX_SI, BX_DI, BP_SI, BP_DI, -- 8086/80186/80286
-- combinations
ST, ST1, ST2, ST3, -- floating registers
-- (stack)
ST4, ST5, ST6, ST7,
nil);

-- the extended registers (EAX .. EDI) plus FS and GS
-- are only allowed in 80386 targets

type scale_type is (scale_1, scale_2, scale_4, scale_8);

subtype machine_string is string(1..100);

pragma page;
type machine_instruction (operand_kind : operand_type) is
record
opcode : opcode_type;

case operand_kind is
when immediate =>
immediat1 : integer; -- immediate

when register =>
r_register : register_type;
-- source and/or destination

when address =>
a_segment : register_type;
-- source and/or destination
a_address_base : register_type;
a_address_index : register_type;
a_address_scale : scale_type;
a_address_offset : integer;

when system_address =>
sa_address : system.address; -- destination

when name =>
n_string : machine_string; -- CALL destination

when register_immediate =>
r_i_register_to : register_type;
-- destination
r_i_immediate : integer;
-- source

when register_register =>
r_r_register_to : register_type;
-- destination
r_r_register_from : register_type;

```

## User's Guide

### Implementation-Dependent Characteristics

```
-- source

when register_address =>
  r_a register_to : register_type;
  -- destination
  r_a segment    : register_type;
  -- source
  r_a address_base : register_type;
  r_a address_index : register_type;
  r_a address_scale : scale_type;
  r_a address_offset : integer;

when address_register =>
  a_r segment : register_type;
  -- destination
  a_r address_base : register_type;
  a_r address_index : register_type;
  a_r address_scale : scale_type;
  a_r address_offset : integer;
  a_r register_from : register_type;
  -- source

when register_system_address =>
  r_sa register_to : register_type;
  -- destination
  r_sa address : system.address;
  -- source

when system_address_register =>
  sa_r address : system.address;
  -- destination
  sa_r reg_from : register_type;
  -- source

when address_immediate =>
  a_i segment : register_type;
  -- destination
  a_i address_base : register_type;
  a_i address_index : register_type;
  a_i address_scale : scale_type;
  a_i address_offset : integer;
  a_i immediate : integer;
  -- source

when system_address_immediate =>
  sa_i address : system.address;
  -- destination
  sa_i immediate : integer;
  -- source

when immediate_register =>
  i_r immediate : integer;
  -- destination
  i_r register : register_type;
  -- source

when immediate_immediate =>
  i_i immediate1 : integer;
  -- Immediate1
  i_i immediate2 : integer;
  -- Immediate2

when register_register_immediate =>
  r_r i_register1 : register_type;
  -- destination
  r_r i_register2 : register_type;
  -- source1
  r_r i_immediate : integer;
  -- source2

when register_address_immediate =>
  r_a i_register : register_type;
  -- destination
  r_a i_segment : register_type;
```



## User's Guide Implementation-Dependent Characteristics

```

-- source1
r_a_i_address_base : register_type;
r_a_i_address_index : register_type;
r_a_i_address_scale : scale_type;
r_a_i_address_offset : integer;
r_a_i_immediate : integer;
-- source2

when register_system_address_immediate =>
  r_sa_i_register : register_type;
  -- destination
  addr10 : system.address;
  -- source1
  r_sa_i_immediate : integer;
  -- source2

when address_register_immediate =>
  a_r_i_segment : register_type;
  -- destination
  a_r_i_address_base : register_type;
  a_r_i_address_index : register_type;
  a_r_i_address_scale : scale_type;
  a_r_i_address_offset : integer;
  a_r_i_register : register_type;
  -- source1
  a_r_i_immediate : integer;
  -- source2

when system_address_register_immediate =>
  sa_r_i_address : system.address;
  -- destination
  sa_r_i_register : register_type;
  -- source1
  sa_r_i_immediate : integer;
  -- source2

when others =>
  null;
end case;
end record;

end machine_code;

```

## 2. Restrictions

Only procedures, and not functions, may contain machine code insertions.

Symbolic names in the form `x'ADDRESS` can only be used in the following cases:

1. `x` is an object of scalar type or access type declared as an object, a formal parameter, or by static renaming.
2. `x` is an array with static constraints declared as an object (not as a formal parameter or by renaming).
3. `x` is a record declared as an object (not a formal parameter or by renaming).

The `m_CALL` can be used with "name" to call (for) a routine.

Two opcodes to handle labels have been defined:

## User's Guide

### Implementation-Dependent Characteristics

- m\_label:** defines a label. The label number must be in the range  $1 \leq x \leq 999$  and is put in the offset field in the first operand of the `MACHINE_INSTRUCTION`.
- m\_reset:** used to enable use of more than 999 labels. The label number after a `m_RESET` must be in the range  $1 \leq x \leq 999$ . To avoid errors you must make sure that all used labels have been defined before a reset, since the reset operation clears all used labels.

All floating instructions have at most one operand which can be any of the following:

- a memory address
- a register or an immediate value
- an entry in the floating stack

### 3. Examples

The following section contains examples of how to use the machine code insertions and lists the generated code.

### 4. Example Using Labels

The following assembler code can be described by machine code insertions as shown:

```
MOV AX, 7
MOV CX, 4
CMP AX, CX
JG 1
JE 2
MOV CX, AX
1: ADD AX, CX
2: MOV SS: [BP+DI], AX

package example_MC is
    procedure test_labels;
    pragma inline (test_labels);
end example_MC;

with MACHINE_CODE; use MACHINE_CODE;
package body example_MC is

    procedure test_labels is
    begin
        MACHINE_INSTRUCTION' (register_immediate, m_MOV, AX, 7);
        MACHINE_INSTRUCTION' (register_immediate, m_MOV, CX, 4);
        MACHINE_INSTRUCTION' (register_register, m_CMP, AX, CX);
        MACHINE_INSTRUCTION' (immediate, m_JG, 1);
        MACHINE_INSTRUCTION' (immediate, m_JE, 2);
        MACHINE_INSTRUCTION' (register_register, m_MOV, CX, AX);
        MACHINE_INSTRUCTION' (immediate, m_label, 1);
        MACHINE_INSTRUCTION' (register_register, m_ADD, AX, CX);
        MACHINE_INSTRUCTION' (immediate, m_label, 2);
        MACHINE_INSTRUCTION' (address_register, m_MOV, SS, BP,
            DI, scale_1, 0, AX);
```

## User's Guide

### Implementation-Dependent Characteristics

```
end test_labels;  
end example_MC;
```

#### 5. Advanced Topics

This section describes some of the more intricate details of the workings of the machine code insertion facility. Special attention is paid to the way the Ada objects are referenced in the machine code body, and various alternatives are shown.

##### a. Address Specifications

Package `MACHINE_CODE` provides two alternative ways of specifying an address for an instruction. The first way is referred to as `SYSTEM_ADDRESS` and the parameter associated this one must be specified via `OBJECT'ADDRESS` in the actual `MACHINE_CODE` insertion. The second way closely relates to the addressing which the 80x86 machines employ: an address has the general form

segment:[base+index\*scale+offset]

The `ADDRESS` type expects the machine insertion to contain values for ALL these fields. The default value `NIL` for segment, base, and index may be selected (however, if base is `NIL`, so should index be). Scale **MUST** always be specified as `scale_1`, `scale_2`, `scale_4`, or `scale_8`. For 16 bit targets, `scale_1` is the only legal scale choice. The offset value must be in the range of -32768 .. 32767.

##### b. Referencing Procedure Parameters

The parameters of the procedure that consists of machine code insertions may be referenced by the machine insertions using the `SYSTEM_ADDRESS` or `ADDRESS` formats explained above. However, there is a great difference in the way in which they may be specified; whether the procedure is specified as `INLINE` or not.

`INLINE` machine insertions can deal with the parameters (and other visible variables) using the `SYSTEM_ADDRESS` form. This will be dealt with correctly even if the actual values are constants. Using the `ADDRESS` form in this context will be the user's responsibility since the user obviously attempts to address using register values obtained via other machine insertions. It is in general not possible to load the address of a parameter because an 'address' is a two component structure (selector and offset), and the only instruction to load an immediate address is the `LEA`, which will only give the offset. If coding requires access to addresses like this, one cannot `INLINE` expand the machine insertions. Care should be taken with references to objects outside the current block since the code generator in order to calculate the proper frame value (using the display in each frame) will apply extra registers. The parameter addresses will, however, be calculated at the entry to the `INLINE` expanded routine to minimize this problem. `INLINE` expanded routines should **NOT** employ any `RET` instructions.

## User's Guide

### Implementation-Dependent Characteristics

Pure procedure machine insertions need to know the layout of the parameters presented to, in this case, the called procedure. In particular, careful knowledge about the way parameters are passed is required to achieve a successful machine procedure. Again there are two alternatives:

The first assumes that the user takes over the responsibility for parameter addressing. With this method, the `SYSTEM_ADDRESS` format does not make sense (since it expects a procedural setup that is not set up in a machine procedure). The user must code the exit from the procedure and is also responsible for taking off parameters if so is required. The rules of Ada procedure calls must be followed. The calling conventions are summarized below.

The second alternative assumes that a specific abstract A-code insertion is present in the beginning and end of the machine procedure. Abstract A-code insertions are not generally available to an Ada user since they require extensive knowledge about the compiler intermediate text called abstract A-code. Thus, they will not be explained further here except for the below use.

These insertions enable the user to setup the procedural frame as expected by Ada and then allow the form `SYSTEM_ADDRESS` in accesses to parameters and variables. Again it is required to know the calling conventions to some extent; mainly to the extent that the access method for variables is clear. A record is, for example, transferred via its address, so access to record fields must first employ an `LES`-instruction and then use `ADDRESS` form using the read registers.

The insertions to apply in the beginning are:

```
pragma abstract_acode_insertions(true);
aa_instr'(aa_Create_Block,x,y,0,0,0);
aa_instr'(aa_End_of_declpart,0,0,0,0,0);
pragma abstract_acode_insertions(false);
```

and at the end:

```
pragma abstract_acode_insertions(true);
aa_instr'(aa_Exit_subprgrm,x,0,x,nil_arg,nil_arg); -- (1)
aa_instr'(aa_Set_block_level,y-1,0,0,0,0);
pragma abstract_acode_insertions(false);
```

where the `x` value represents the number of words taken by the parameters, and `y` is the lexical block level of the machine procedure. However, if the procedure should leave the parameters on the stack (scalar `IN` `OUT` or `OUT` parameters), then the `Exit_subprgrm` insertion should read:

```
aa_instr'(aa_Exit_subprgrm,0,0,0,nil_arg,nil_arg); -- (2)
```

In this case, the caller moves the updated scalar values from the stack to their destinations after the call.

The `NIL_ARG` should be defined as :

User's Guide  
Implementation-Dependent Characteristics

`nil_arg : constant := -32768;`

**WARNING:** When using the `AA_INSTR` insertions, great care must be taken to assure that the `x` and `y` values are specified correctly. Failure to do this may lead to unpredictable crashes in compiler pass8.

**c. Parameter Transfer**

It may be a problem to figure out the correct number of words which the parameters take up on the stack (the `x` value). The following is a short description of the transfer method:

**INTEGER** types take up at least 1 storage unit. 32 bit integer types take up 2 words, and 64 bit integer types take up 4 words. In 32 bit targets, 16 bit integer types take up 2 words the low word being the value and the high word being an alignment word. **TASKs** are transferred as **INTEGER**.

**ENUMERATION** types take up as 16 bit **INTEGER** types (see above).

**FLOAT** types take up 2 words for 32 bit floats and 4 words for 64 bit floats.

**ACCESS** types are considered scalar values and consist of a 16 bit segment value and a 16 or 32 bit offset value. When 32 bit offset value, the segment value takes up 2 words the high word being the alignment word. The offset word(s) are the lowest, and the segment word(s) are the highest.

**RECORD** types are always transferred by address. A record is never a scalar value (so no post-procedure action is carried out when the record parameter is `OUT` or `IN OUT`). The representation is as for **ACCESS** types.

**ARRAY** values are transferred as one or two **ACCESS** values. If the array is constrained, only the array data address is transferred in the same manner as an **ACCESS** value. If the array is unconstrained below, the data address will be pushed by the address of the constraint. In this case, the two **ACCESS** values will **NOT** have any alignment words in 32 bit targets.

**Packed ARRAY** values (e.g. **STRING** types) are transferred as **ARRAY** values with the addition of an **INTEGER** bit offset as the highest word(s):

```
+H: BIT_OFFSET
+L: DATA_ADDRESS
+0: CONSTRAINT_ADDRESS    -- may be missing
```

The values `L` and `H` depend on the presence/absence of the constraint address and the sizes of constraint and data addresses.

In the two latter cases, the form parameter's address will always yield the address of the data. If access is required to constraint or bit offset, the instructions must use the **ADDRESS** form.

## User's Guide Implementation-Dependent Characteristics

### d. Example

A small example is shown below (16 bit target):

```
procedure unsigned_add
```

```
    (op1  : in    integer;
     op2   : in    integer;
     res   : out   integer);
```

Notice that machine subprograms cannot be functions.

The parameters take up:

op1	: integer	:	1 word
op2	: integer	:	1 word
res	: integer	:	1 word
Total	:		3 words

The body of the procedure might then be the following assuming that the procedure is defined at outermost package level:

```
procedure unsigned_add
    (op1  : in    integer;
     op2   : in    integer;
     res   : out   integer) is
begin
    pragma abstract_acode_insertions(true);
    aa_instr'(aa_Create_Block,3,1,0,0,0); -- x = 3, y = 1
    aa_instr'(aa_End_of_declpart,0,0,0,0,0);
    pragma abstract_acode_insertions(false);

    machine_instruction'(register_system_address, m_MOV,
                        AX, op1'address);
    machine_instruction'(register_system_address, m_ADD,
                        AX, op2'address);
    machine_instruction'(immediate, m_JNC, 1);
    machine_instruction'(immediate, m_INT, 5);
    machine_instruction'(immediate, m_label, 1);
    machine_instruction'(system_address_register, m_MOV,
                        res'address, AX);

    pragma abstract_acode_insertions(true);
    aa_instr'(aa_Exit_subprgm,0,0,0,nil_arg); -- (2)
    aa_instr'(aa_Set_Block_level,0,0,0,0,0); -- y-1 = 0
    pragma abstract_acode_insertions(false);
end unsigned_add;
```

A routine of this complexity is a candidate for `INLINE` expansion. In this case, no changes to the above 'machine\_instruction' statements are required. Please notice that there is a difference between addressing record fields when the routine is `INLINE` and when it is not:

## User's Guide Implementation-Dependent Characteristics

```
type rec is
  record
    low      : integer;
    high     : integer;
  end record;
```

```
procedure add_32 is
  (op1 : in integer;
   op2 : in integer;
   res  : out rec);
```

The parameters take up  $1 + 1 + 2$  words = 4 words. The RES parameter will be addressed directly when `INLINE` expanded, i.e. it is possible to write:

```
machine_instruction'(system_address_register, m_MOV,
                    res'address, AX);
```

This would, in the not `INLINED` version, be the same as updating that place on the stack where the address of RES is placed. In this case, the insertion must read:

```
machine_instruction'(register_system_address, m_LES,
                    SI, res'address);
-- LES SI,[BP+...]
machine_instruction'(address_register, m_MOV,
                    ES, SI, nil, scale_1, 0, AX);
-- MOV ES:[SI+0],AX
```

As may be seen, great care must be taken to ensure correct machine code insertions. A help could be to first write the routine in Ada, then disassemble to see the involved addressings, and finally write the machine procedure using the collected knowledge.

Please notice that `INLINED` machine insertions also generate code for the procedure itself. This code will be removed when the `/NOCHECK` qualifier is applied to the compilation. Also not `INLINED` procedures using the `AA_INSTR` insertion, which is explained above, will automatically get a `storage_check` call (as do all Ada subprograms). On top of that, 8 bytes are set aside in the created frame, which may freely be used by the routine as temporary space. The 8 bytes are located just below the display vector of the frame (from SP and up). The `storage_check` call will not be generated when the compiler is invoked with `/NOCHECK`.

The user also has the option NOT to create any blocks at all, but then he should be certain that the return from the routine is made in the proper way (use the RETP instruction (return and pop) or the RET). Again it will help first to do an Ada version and see what the compiler expects to be done.

Symbolic fixups are possible in certain instructions. With these you may build 'symbolic' instructions byte for byte. The instructions involved all require the operand type NAME (like used with CALL), and the interpretation is the following:

**User's Guide**  
**Implementation-Dependent Characteristics**

<b>(name, m_DATAD, "MYNAME")</b>	<b>a full virtual address (offset and selector) of the symbol MYNAME (no additional offset is possible).</b>
<b>(name, m_DATAW, "MYNAME")</b>	<b>the offset part of the symbol MYNAME (no additional offset is possible).</b>
<b>(name, m_DATAB, "MYNAME")</b>	<b>the selector value of symbol MYNAME</b>

**In inlined machine instructions it may be a problem to obtain the address of a parameter (rather than the value). The LEA instruction may be used to get the offset part, but now the following form allows a way to load a selector value as well:**

**(system\_address, LES, param'address)** **ES is loaded with the selector of PARAM. If this selector was e.g. SS, it would be pushed and popped into ES. LES may be substituted for LFS and LGS for 80386.**



APPENDIX F  
PART II -

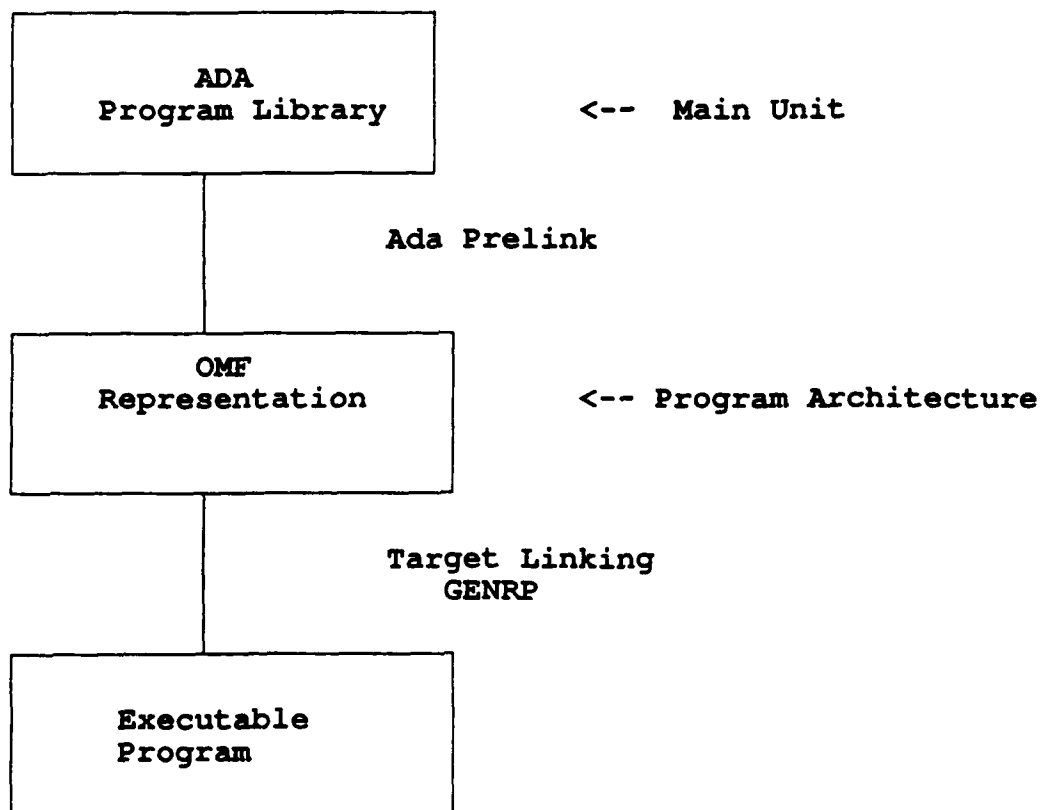
TARGET-  
DEPENDENT  
CHARACTERISTICS

1 Ada Linker - [LIN]

[User\_Guide] lists the capabilities of Ada linker with respect to Ada programs programming aspects and to language definitions meaning. This section integrates DDC Ada Compiler System User Guide with respect to Ada linker use to generate Ada programs for Alenia computers. The reader is expected to be familiar with the terminology of Alenia computers Programming and Executing Environments ([SEL1 87], [SEL2 87]), as well as with Ada terminology ([LRM 83] and [User\_guide]).

1.1 Linking Process

Ada linking process can be described as in the following figure.



Horizontal arrows represent inputs to linking process, while vertical arrows represent actions performed by the linker consequent on those inputs.

Two phases are identified in the linking process.

The former phase produces Ada program Intel Object Module Format (OMF) representation. One or more host files support such a representation. This is the collection of Ada compilation units selected by <main unit>program, and correspond to Ada program as in [LRM 83] chapter 10.

In the latter phase Alenia Software Factory works to integrate OMF Ada program with the piece of information to obtain an executable program for Alenia computer. All Ada Run Time Supports are introduced during this phase.

Ada Linker can be executed in two different modes, procedural and automatic.

Procedural mode consists in successive and explicit calls to the various tools necessary to generate the executable program.

Therefore, it provides call to Ada prelink, to linker target and to GENRP. Call to Ada prelink occurs through / PRELINK qualifier (par. 1.1.1).

The automatic condition to generate an executable program employs /LINK qualifier (par. 1.1.2) to activate automatically, in cascade, all tools used.

Beside the qualifiers described in [User\_Guide] and [CLU], the following qualifiers are allowed:

```
/PRELINK;  
/LINK;  
/TEMPLATE;  
/MAP;  
/DEBUG;
```

/PRELINK and /LINK accept all the qualifiers described in "ADA Compiler User Guide" Cap.6.

/LINK also accept /TEMPLATE, /MAP, /DEBUG.

#### 1.1.1 /PRELINK qualifier -

/PRELINK qualifier, defined by installation ADA 'command verb', activates Ada prelink (it must be used in the place of /LINK

qualifier in [User\_Guide] chap.6, therefore refer for use to such chapter).

To generate an Ada program obtained with the procedural mode, operations listed below must be implemented:

(be 'ada' the command verb used for Ada software factory installation in host environment;

be 'main' the Ada unit name compiled in 'Ada\_library';

be 'graph.gra' the name of a graph;

be 'template' the name of a program architecture given in graph.gra )

- 1) - ada/prelink/library='ada\_library'/ofd=[] <main>
- 2) - @<main>\_link
- 3) - genrp template,graph.gra

1) The command

ada/prelink/library='ada\_library'/ofd=[] <main>

activates Ada prelink.

In input it takes:

- The current sublibrary which includes the compiled <main> unit
- Object File Directory name (OFD = [...])
- Main unit name

In output it produces:

- An assembler file called <main>\_elab.asm which declares a procedure called CG?ADAMAINPROGRAM, that is also known as "anonymous task". It defines the processing order of main context (packages and subprogram of which a "with" clause has been done), and activates the main itself.

- A command file called <main>\_LINK.COM which invokes the ASSEMBLER on <main>\_elab.asm, generates a temporary file LINK.CTF which includes all object files that must be linked to the "main program" and invokes the BINDER286 by trasmitting to it in input, LINK.CTF file. BINDER286 output is a file whose name is the same as "main unit" name, and has .LNK extention,

2) The command

@<main>\_LINK

executes <main>\_LINK.COM file produced in the previous phase.

3) The command

GENRP 'template', 'graph.gra'

invokes GENRP [ref. MARA286 Computer Manual].

NOTE: If the files containing useful information for debugging (symbol file and map in Ada format) are to be produced, it is necessary:

- To define "MAPPER\_LIBRARY" logic on "ADA\_LIBRARY" (logic ex: DEFINE Mapper\_library ada286\_library).

- To specify ADAGEN in GENRP command (ex: GENRP 'template', 'graph.gra' ADAGEN)

An example of the template to be used with the procedural mode follows.

• Example

Be MYPROC the name of Ada main unit.

Be MYPROC.LNK the name of the output file produced by LINK.COM.

An example of program template follows:

```
system mysys;

program template mytemp large
    code PRIVATE
    data PRIVATE NODAL;

module :FMS:ADARTSA1/RTSDATA.OBJ      relocatable;
module :FMS:ADABIOA1/BIODATA.OBJ      relocatable;
module MYPROC.LINK                    relocatable;
module :FMS:DACS86A0/ROOT.LIB          relocatable;
module :FMS:DACS86A0/RTHELP286.LIB     relocatable;
module :FMS:ADABIOA1/BINDER286.LIB     relocatable;
module :FMS:ADARTSA1/BINDER286.LIB     relocatable;
module :FMS:KER286A0/ADAUS.LIB         relocatable;
module GATELBA                        relocatable;

initial procedure MYPROC
stacksegment nodal size = 0FF00H;

end program;

hardware configuration;
volume    DEFAULT_VOLUME,
presence  NODAL origin FIRST NODAL PAGE
          LOCAL to 0 origin FIRST_LOCAL_PAGE;

end configuration;
```

```
include MYTEMP;
initial process on 0 priority 2;

end system;
```

### 1.1.2 LINK qualifier

/LINK qualifier, defined by installation ADA 'command verb', enables to obtain an executable program (.LTL) through a unique command in which appropriate qualifiers specify (see [User's Guide]) the parameters to be passed to the various tools involved in the generation.

An example of invocation of automatic linking process is given below:

```
(be 'ada' the command verb used for Ada software
factory installation in host environment;
```

```
be 'main' the name of Ada unit compiled
in ada_library;)
```

```
- ada/link/library='ada_library'/ofd=[...] <main>
```

The command executes:

- 1) Ada prelink invocation
- 2) <main> LINK.COM commands file execution
- 3) GENRP invocation

The command stresses GENRP with the following parameters:

- A default graph provided on installation is used as graph.
- From default graph is taken the template which has the same name as the graph, and it is used as template.

DEFAULT graph will be called 'ada'.gra where 'ada' is the installation "command verb".

The installation itself defines an 'ada'\_graph logic on default graph.

Every user who needs particular and characteristic performances of its execution environment (which are not provided by the default graph), can define its own graph, but it must assign 'ada'\_graph logic name to this graph.

Example:

- 1) \$ define ada286\_graph mygra.gra

2) \$ ada286/link/ofd=[]/library=mylib.alb main

These commands implicitly define the following GENRP invocation:

```
$ GENRP mygra, mygra.gra
```

i.e., it is assumed that in mygra.gra graph, a template called mygra exists

[ref. par. 1.1.2.1.1 for additional information]

Default graph provides two program architectures relative to mono and multiprocessor generations.

The graph SCL description is reported below:

```
system <graph_name>;

program template <graph_name> large
    code PRIVATE
    data PRIVATE NODAL;
module :FMS:ADARTSA1/RTSDATA.OBJ      relocatable;
module :FMS:ADABIOA1/BIODATA.OBJ      relocatable;
module <graph_name>                   relocatable;
module :FMS:DACS86A0/ROOT.LIB          relocatable;
module :FMS:DACS86A0/RTHELP286.LIB     relocatable;
module :FMS:ADABIOA1/BINDER286.LIB     relocatable;
module :FMS:ADARTSA1/BINDER286.LIB     relocatable;
module :FMS:KER286A0/ADAUS.LIB         relocatable;
module GATELBA                         relocatable;

initial procedure cg_AdaMainProgram
stacksegment nodal size = 0FF00H;

end program;
$ eject
```

```
program template MULTI <graph_name> large
    code PRIVATE
    data PRIVATE NODAL;

Subprogram GENERAL Repeatable;
module :FMS:ADARTSA1/RTSDATA.OBJ      relocatable;
module :FMS:ADABIOA1/BIODATA.OBJ      relocatable;
module ELABORATION                     source asm;
$include(Repeated)
module :FMS:DACS86A0/ROOT.LIB          relocatable;
module :FMS:DACS86A0/RTHELP286.LIB     relocatable;
module :FMS:ADABIOA1/BINDER286.LIB     relocatable;
module :FMS:ADARTSA1/BINDER286.LIB     relocatable;
module :FMS:KER286A0/ADAUS.LIB         relocatable;
module GATELBA                         relocatable;
```

```

end subprogram;

subprogram ZONE_0 optional;
$include(Zone0)
  module :FMS:DACS86A0/ROOT.LIB          relocatable;
  module :FMS:DACS86A0/RTHELP286.LIB     relocatable;
  module :FMS:ADABIOA1/BINDER286.LIB     relocatable;
  module :FMS:ADARTSA1/BINDER286.LIB     relocatable;
  module :FMS:KER286A0/ADAUS.LIB         relocatable;
  module GATELBA                          relocatable;
end subprogram;

subprogram ZONE_1 optional;
$include(Zone1)
  module :FMS:DACS86A0/ROOT.LIB          relocatable;
  module :FMS:DACS86A0/RTHELP286.LIB     relocatable;
  module :FMS:ADABIOA1/BINDER286.LIB     relocatable;
  module :FMS:ADARTSA1/BINDER286.LIB     relocatable;
  module :FMS:KER286A1/ADAUS.LIB         relocatable;
  module GATELBA                          relocatable;
end subprogram;

subprogram ZONE_7 optional;
$include(Zone7)
  module :FMS:DACS86A0/ROOT.LIB          relocatable;
  module :FMS:DACS86A0/RTHELP286.LIB     relocatable;
  module :FMS:KER286A0/ADAUS.LIB         relocatable;
  module :FMS:ADABIOA1/BINDER286.LIB     relocatable;
  module :FMS:ADARTSA1/BINDER286.LIB     relocatable;
  module :FMS:KER286A0/ADAUS.LIB         relocatable;
  module GATELBA                          relocatable;
end subprogram;

subprogram ZONE_8 optional;
$include(Zone8)
  module :FMS:DACS86A0/ROOT.LIB          relocatable;
  module :FMS:DACS86A0/RTHELP286.LIB     relocatable;
  module :FMS:ADABIOA1/BINDER286.LIB     relocatable;
  module :FMS:ADARTSA1/BINDER286.LIB     relocatable;
  module :FMS:KER286A0/ADAUS.LIB         relocatable;
  module GATELBA                          relocatable;
end subprogram;

subprogram ZONE_9 optional;
$include(Zone9)
  module :FMS:DACS86A0/ROOT.LIB          relocatable;
  module :FMS:DACS86A0/RTHELP286.LIB     relocatable;
  module :FMS:ADARTSA1/BINDER286.LIB     relocatable;
  module :FMS:ADABIOA1/BINDER286.LIB     relocatable;
  module :FMS:KER286A0/ADAUS.LIB         relocatable;
  module GATELBA                          relocatable;
end subprogram;

subprogram ZONE_10 optional;
$include(Zone10)
  module :FMS:DACS86A0/ROOT.LIB          relocatable;

```

```

module :FMS:DACS86A0/RTHelp286.LIB      relocatable;
module :FMS:ADABIOA1/BINDER286.LIB      relocatable;
module :FMS:ADARTSA1/BINDER286.LIB      relocatable;
module :FMS:KER286A0/ADAUS.LIB           relocatable;
module GATELBA                           relocatable;
end subprogram;

subprogram ZONE_11 optional;
$include(Zone11)
module :FMS:DACS86A0/ROOT.LIB            relocatable;
module :FMS:DACS86A0/RTHelp286.LIB      relocatable;
module :FMS:ADABIOA1/BINDER286.LIB      relocatable;
module :FMS:ADARTSA1/BINDER286.LIB      relocatable;
module :FMS:KER286A0/ADAUS.LIB           relocatable;
module GATELBA                           relocatable;
end subprogram;

subprogram ZONE_12 optional;
$include(Zone12)
module :FMS:DACS86A0/ROOT.LIB            relocatable;
module :FMS:DACS86A0/RTHelp286.LIB      relocatable;
module :FMS:ADABIOA1/BINDER286.LIB      relocatable;
module :FMS:ADARTSA1/BINDER286.LIB      relocatable;
module :FMS:KER286A0/ADAUS.LIB           relocatable;
module GATELBA                           relocatable;
end subprogram;

subprogram ZONE_13 optional;
$include(Zone13)
module :FMS:DACS86A0/ROOT.LIB            relocatable;
module :FMS:DACS86A0/RTHelp286.LIB      relocatable;
module :FMS:ADABIOA1/BINDER286.LIB      relocatable;
module :FMS:ADARTSA1/BINDER286.LIB      relocatable;
module :FMS:KER286A0/ADAUS.LIB           relocatable;
module GATELBA                           relocatable;
end subprogram;

subprogram ZONE_14 optional;
$include(Zone14)
module :FMS:DACS86A0/ROOT.LIB            relocatable;
module :FMS:DACS86A0/RTHelp286.LIB      relocatable;
module :FMS:ADABIOA1/BINDER286.LIB      relocatable;
module :FMS:ADARTSA1/BINDER286.LIB      relocatable;
module :FMS:KER286A0/ADAUS.LIB           relocatable;
module GATELBA                           relocatable;
end subprogram;

subprogram ZONE_15 optional;
$include(Zone15)
module :FMS:DACS86A0/ROOT.LIB            relocatable;
module :FMS:DACS86A0/RTHelp286.LIB      relocatable;
module :FMS:ADABIOA1/BINDER286.LIB      relocatable;
module :FMS:ADARTSA1/BINDER286.LIB      relocatable;
module :FMS:KER286A0/ADAUS.LIB           relocatable;
module GATELBA                           relocatable;
end subprogram;

```



```
initial procedure CG_AdaMainProgram
stacksegment nodal size = 0ff00h;
```

```
end program;
hardware configuration;
```

```
    volume DEFAULT_VOLUME,
    presence    NODAL origin FIRST_NODAL_PAGE
LOCAL to 0 origin FIRST_LOCAL_PAGE,
LOCAL to 1 origin FIRST_LOCAL_PAGE,
LOCAL to 2 origin FIRST_LOCAL_PAGE,
LOCAL to 3 origin FIRST_LOCAL_PAGE,
LOCAL to 4 origin FIRST_LOCAL_PAGE,
LOCAL to 5 origin FIRST_LOCAL_PAGE,
LOCAL to 6 origin FIRST_LOCAL_PAGE,
LOCAL to 7 origin FIRST_LOCAL_PAGE,
LOCAL to 8 origin FIRST_LOCAL_PAGE,
LOCAL to 9 origin FIRST_LOCAL_PAGE,
LOCAL to 10 origin FIRST_LOCAL_PAGE,
LOCAL to 11 origin FIRST_LOCAL_PAGE,
LOCAL to 12 origin FIRST_LOCAL_PAGE,
LOCAL to 13 origin FIRST_LOCAL_PAGE,
LOCAL to 14 origin FIRST_LOCAL_PAGE,
LOCAL to 15 origin FIRST_LOCAL_PAGE;
end configuration;
```

```
include <graph_name>;
initial process on 0 priority 2;
```

```
include MULTI_<graph_name>;
$include(SubprogramAssign)
initial process on 0 priority 2;
```

```
end system;
```

#### 1.1.2.1 /TEMPLATE qualifier

<template> ::= /TEMPLATE [ = <identifier> ]

<template> allows to specify which program architecture in the <graph> file specified by 'ada'\_graph, is required to generate an executable program.

The following policies apply when <template> is either missing or partially or totally specified.

- a) if <template> is missing, then the following default is valid:  
/template = <graph\_name>
- b) if only /template is specified, then the following default is valid:  
/template = <main>

c) if /template = <name> is present, we assume that <name> is a program in the graph indicated by <'ada'\_graph>

A GENRP error is reported when none of the above conditions can be matched.

For more details see Par. 1.1.2.1.1 and 1.1.3.

#### 1.1.2.1.1 WRITING A PROGRAM TEMPLATE

Some precautions on writing a program template allows to make it "universal" i.e. it can be used with various Ada programs with no modifications.

In par. 1.1.1 an example of program template specialized in an Ada procedure called MYPROC has already been illustrated.

It is clear that the produced graph can be exclusively used with procedures which have the same name (MYPROC.LNK).

However, it is just by utilizing Ada linker structure that is possible to reach that program template "universality" of use mentioned above.

In order to reach this universality, it is necessary:

1) To assign the same name to the template and to the main program included in it.

• Example:

Program template STANDARD LARGE

CODE PRIVATE  
DATA PRIVATE NODAL

```
module :FMS:ADARTSA1/RTSDATA.OBJ relocatable;
module :FMS:ADABIOA1/BIODATA.OBJ relocatable;
module STANDARD
module :FMS:DACS86A0/ROOT.I-IB          relocatable;
module :FMS:DACS86A0/RTHelp286.LIB      relocatable;
module :FMS:ADABIOA1/BINDER286.LIB      relocatable;
module :FMS:ADARTSA1/BINDER286.LIB      relocatable;
module :FMS:KER286A1/ADAUS.LIB           relocatable;
module GATELBA                           relocatable;
```

STANDARD is considered by Ada linker as a logic name associated to the last .LNK file produced.

2) To assign to initial procedure CG\_ADAMAINPROGRAM name exported by Ada linker as entry point of the anonymous task.

• Example:

Initial procedure CG\_ADAMAINPROGRAM  
stack segment nodal size = 0FF00H

These two rules together with 'ada' graph logic definition, and the automatic use of Ada linker [rif. 1.1.2] make very easy the generation of a relocatable program.

In addition, a graph construction will be done once at the beginning of an application design.

This will allow the user to focus more on those aspects of Ada language concerning its application.

#### 1.1.2.2 /MAP qualifier

/NOMAP (default)

It is necessary to create the map (<main>.MGA) of the program to be generated.

As regards map format and further information, see Mapper Userguide [ADAMAP] and [GRP].

#### 1.1.2.3 /DEBUG Qualifier -

/NODEBUG (default)

It is necessary for program debugging through IDA286 symbolic debugger.

The output produced consists of 2 files:

- <main>.SYM
- <main>.TLD

(see IDA286 User's Guide)

#### 1.1.3 Example of Linker Use -

Generation of Ada programs on a monoprocessor.

Let's suppose that CATRIN.SCL contains the architectural description of our execution environment.

scl text:

system CATRIN;

```
    program template CATRIN . . .  
    end program;  
    program template IPL_CATRIN . . .  
    end program;  
    program template TEST_PROGRAM . . .  
    end program;
```

...

end system;

Be SOME\_DIRECTORY the one in which CATRIN.GRA graph is produced.

Now let's assume that following Ada text refers to an Ada main program:

ada text:

```
    procedure TEST_PROGRAM is . . .  
    begin . . .  
    end;
```

It is necessary to define SOME\_DIRECTORY:CATRIN.GRA as the user default graph: ( ADA286 = Ada Command verb )

\$ define ada286\_graph some\_directory:catrin.gra

Suppose that the linker invocation command for Ada TEST\_PROGRAM procedural generation is the following:

```
ada286/prelink/ofd=[]/lib=.../template=CATRIN TEST_PROGRAM  
  
@<main>_link  
  
genrp CATRIN,ADA286_GRAPH
```

where CATRIN is the template used for the generation and CATRIN.GRA is the graph.file.

Linker invocation command for Ada TEST\_PROGRAM automatic generation can be one of the following:

- a) ada286/link/ofd=[]/lib=.../template=IPL\_CATRIN TEST\_PROGRAM
- b) ada286/link/ofd=[]/lib=.../template TEST\_PROGRAM
- c) ada286/link/ofd=[]/lib=... TEST\_PROGRAM

The following generation rules apply to a), b) and c) [ ref. par. 1.1.2.1 ]:

- a) IPL\_CATRIN template of CATRIN\_GRA is used for generation
- b) TEST\_PROGRAM template of CATRIN\_GRA is used for generation
- c) CATRIN\_TEMPLATE of CATRIN\_GRA is used for generation

## 2 ADA LIBRARY

The [User Guide] reports organization and supporting tools of the DDC 80x86 Cross Compiler System Program Library.

Here is the detailed structure of Ada program library delivered in DACS86 product. Ada specification and comments of the program units compiled in the Root Level Sublibrary are also included.

The root level of program library is subdued to configuration management, it owns a version and a release number. It is reserved for the compilation of general purpose services which range from language standard domain to implementation standard domain. Therefore, the user is strongly discouraged to compile in this sublibrary.

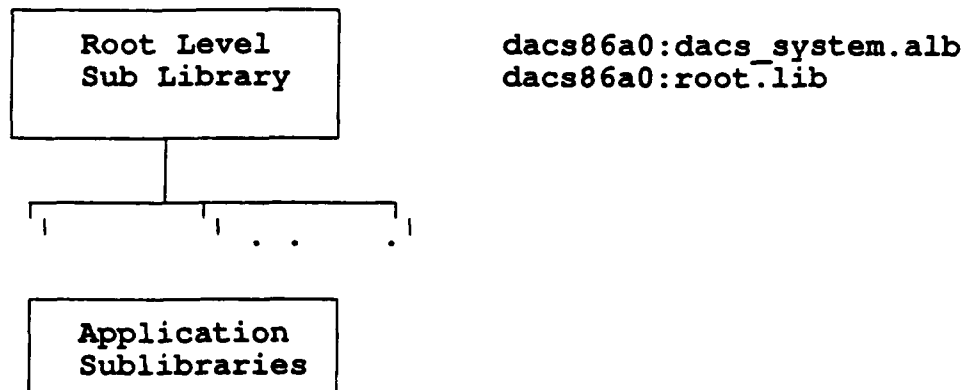
User compilations are recommended to introduce Program Units into the Program Library starting from a second level sublibrary. Refer to [User Guide] for details about sublibraries creation.

The first section reports the organization and the instruction for use of delivered program library.

The following two sections describe organization and contents, in terms of program units, of Ada Program Library delivered by the implementation.

### 2.1 Program library

The figure represents DACS 80x86 VAX/VMS crossed Ada Program Library delivered by the Implementation.



VAX/VMS files indicate the host files representing the sublibrary.

DACS86A0:DACS\_SYSTEM.ALB collects all program units delivered (compiled) in the root sublibrary. To make the extraction of Ada application from the library more efficient, the corresponding object files are collected in Intel LIB286 library DACS86A0:ROOT.LIB, too. These two files are therefore strongly matched to each other, and the user should consider this kind of consistency when he decides to perform any compilation in root sublibrary. In chapter 7 of [User's Guide] can be found instructions necessary to update DACS86A0:ROOT.LIB file when new compilations are added to the root.

## 2.2 Root Sublibrary Content

Root Level Sublibrary exports services defined by the predefined language packages and subprograms, as indicated in package STANDARD declaration (cfr. [LRM] Appendix C). Specifications and relative items depending on the implementation can be found in Appendix F of [User\_Guide].

This section also introduces the user to Ada supports defined to access to the capabilities of Alenia computers architectural components, both hardware and software.

Most of the above components present an environmental interface, which is a library, may be generic, package specification. This means that an application program can use them through the context clause, as illustrated in the following example.

Example:

```
with <some_architectural_components> ;  
procedure <some_application> is ... end;
```

In the library are delivered some packages necessary to support the Cross Compiler and the Run Time Support.

### 3 MULTIPROCESSOR PROGRAM GENERATION - [MPG]

Ada toolset supports multiprocessor target architectures, allowing the code distribution to private memories of various processors, and the possibility at run Time - through a particular package - to fix the processor which must perform a particular task (ref. ADA RTS package in [APP B]).

The user who prepares to implement a multiprocessor Ada program must assure the physical addressability of objects and agents which, though Ada visible, may be not physically visible, as they are wrongly assigned in node memory.

Examples:

- Code addressability

```
package body P1 is  
...  
...  
    X:integer  
    procedure SOME_PROCEDURE(Y:in integer) is separate;  
...  
...  
begin  
    ...  
    ...  
    SOME_PROCEDURE(X)  
    ...  
    ...  
end;
```

In order to allow the processing of this Ada text, SOME\_PROCEDURE compilation unit must have been allocated on the same processor which is processing the P1 compilation unit.

- Data addressability

```
package SHARED is  
    VAR_COM: integer := 0;  
end SHARED;  
package body SHARED is . . .  
end SHARED;  
  
with SHARED;  
package TASKDEF is
```

```

        task type T is
            entry E;
        end T;

    end TASKDEF;
    package body TASKDEF is
        task body T is
            begin
                accept E do
                    SHARED.VAR_COM = SHARED.VAR_COM + 1;
                end;
            end T;

        begin . . .
    end TASKDEF;

```

Each EXAMPLE A, EXAMPLE B and EXAMPLE C compilation units include a procedure declaration which, in turn, allocates a T object.

```

C.U.      : EXAMPLE_A
           with TASKDEF; use TASKDEF;
           procedure EXAMPLE_A is
               TASK_A: T;
               ..
               ..
           begin
               ..
               ..
           end EXAMPLE_A;

```

```

C.U.      : EXAMPLE_B
           with TASKDEF; use TASKDEF;
           procedure EXAMPLE_B is
               TASK_B: T;
               ..
               ..
           begin
               ..
               ..
           end EXAMPLE_B;

```

```

C.U.      : EXAMPLE_C
           with TASKDEF; use TASKDEF;
           procedure EXAMPLE_C is
               TASK_C: T;
               ..
               ..
           begin
               ..
               ..
           end EXAMPLE_C;

```

In case EXAMPLE A, EXAMPLE B, and EXAMPLE C procedures are allocated on different processors, their invocation causes the



activation of tasks of T Type on these processors and the need to access to the VAR\_COM shared variable (ref. [LRM] 9.14). This requires a nodal allocation of this variable, in order to make it actually addressable from each task.

Therefore, in order to meet addressability constraints, in relation to code segments, the system designer must be sure that each local area include, within it, the whole compilation unit code imported in this area.

This result can be obtained either by allocating the code - referred from various areas - in the nodal memory, or by repeating this code in local memories of involved processors.

Nodal memory use is absolutely necessary when it is a matter of assuring the addressability of data shared by various tasks on various processors.

Code repetition is advisable in order to assure shared code addressability.

### 3.1 Cluster

SCL configuration language allows the arrangement of a program in subprograms, which make up the allocation unit within the node. Each subprogram, in turn, is composed of one or more relocatable or source format modules.

Particular remarks must be made when Ada language is used.

In fact, this language has a specific tool, the Ada Pre-Linker, which can collect - from the program library - compilation units referred by the main procedure of a particular Ada program. Yet, Ada Pre-linker assures that the processing order (see [LRM ch. 10]) of library units referred by the program is respected, by sequencing its processing in <main>\_ELAB.ASM module. In a multiprocessor context, not all library units are local to the processor which starts the whole program processing. Thus, the migration of program processing is needed when - consistently with the fixed order - the elaboration of units allocated to a processor, different from the current one, is required.

Ada Pre-linker must be informed of what library units have to be processed by what processors, so that these migrations correctly occur.

This piece of information is given in terms of compilation units groups known as CLUSTERS, and to what processors they are allocated. For this purpose, Ada Pre-linker makes available two qualifiers of the command line: /CLUSTER and /PROCESSOR\_ASSIGNMENT (see [CLU]).

Cluster is a collection of compilation units correlated to one another by a logic defined by the user. They represent the allocation unit on a memory area, and as such, they provide the

means to organize an Ada Program in components which - potentially - can be allocated in different memory areas.

### 3.2 Clusters Definition Criteria

This paragraph is aimed to list a series of criteria in order to define the clusters of a specific program which must be processed in a multiprocessor context.

These criteria have an impact on the architectural structure of a multiprocessor program text, implying the need to compile separately the program units which are to be allocated in any cluster. In a monoprocessor context, program units separate compilation is due essentially to reasons relating to software engineering.

In a multiprocessor program design, the identification of those program fragments which must be allocated in different processors becomes particularly important. This either in order to better program performances, or to use particular hardware supports locally connected to these processors.

This identification process is expressed in terms of compilation units grouped in clusters which collect, as mentioned above, the units of allocation to the different processors which are to be involved in the processing.

Thus, the distribution criterium in a multiprocessor hardware context is the leading criterium to clusters definition. We indicate this type of cluster with the term: POLO.

Unlike [SCL] SUBPROGRAMS, clusters are limited by a memory segment physical size. So, for a particular processor, the additional definition of extension clusters can be needed.

Thus, a particular cluster size, in terms of code, can justify additional clusters definition in relation to program poles. We define this type of cluster with the term: GREGARIUS.

Clusters identified by previous criteria, may have non null intersections at the level of the compilation units needed by various processors. It is possible to perform a processor local repetition of these compilation units code without modifying the program logic. However, in order to make this possible, the definition of new clusters with these units is needed to control their repetition.

Thus, the intersection of the poles and relative gregariuses, justifies new clusters introduction. The following term indicates this type of cluster: REPEATER.

The last criterium which can justify the definition of clusters, is the need to put in the same physical memory segment two or more compilation units which often are called. This minimizes the number of run time switchings of code segments in the CS machine register (call NEAR).

Procedural calls efficiency justifies new clusters introduction, or processor clustering reorganization.

We indicate this type of cluster with the term: FAMILY

Besides the justifications already mentioned, there are not other reasons which justify new clusters declaration. This does not mean that these criteria allow the clustering of all compilation units of a program. Rather often the programmer can express for some of them no clusterization of those listed above. Typically, these units lie outside the programmer direct control and are imported in the application by dependence relationships transitively expressed by context clauses.

For these units, a total repetition on all processors involved in the application processing is required. As the support description will show, the user has not to express explicitly this requirement as the tool can define - on its own - an adequate number of clusters that, however, will be shared out to all processors.

### 3.3 SCL Description

Likewise the default architecture for monoprocessor generations, SCL description of multiprocessor generations architecture is the framework of a programs family. Family programs differ in their contents, and for the allocation of 16 optional subprograms, and of a repeatable subprogram.

Subprograms contents parametrization is expressed by using the SCL "\$INCLUDE" directive of a file which lists the compilation units allocated to the relative processor.

Subprograms allocation parametrization is expressed by using the SCL "\$INCLUDE" directive of the file of allocation of non empty subprograms to required processors.

A SCL scheme of a multiprocessor program generic architecture is reported below.

```
system <system_name>;

program template <template_name> large
    <default_code_protection>
    <default_data_protection_and_allocation>;

    <repeatable_subprogram_declaration>;
    { <optional_subprogram_declaration> } 0..15;

    initial procedure <template_name>
        <stack_segment_size_and_allocation>;

end program;

hardware configuration;
    <volume_declaration>
```

```

    <nodal_zone_declaration>
    { <local_zone_declaration> } 0..15
end configuration;
[ <program_activation> ]

end system;

<repeatable_subprogram_declaration> ::=
    subprogram GENERAL repeatable;
    module :FMS:ADARTSA1/RTSDATA.OBJ      relocatable;
    module :FMS:ADABIOA1/BIODATA.OBJ      relocatable;
    module :FMS:ELABORATION                source asm;
    <repeated_list>;
    { <module> };
    module :FMS:DACS86A0/ROOT.LIB          relocatable;
    module :FMS:DACS86A0/RTHelp286.LIB     relocatable;
    module :FMS:ADABIOA1/BINDER286.LIB     relocatable;
    module :FMS:ADARTSA1/BINDER286.LIB     relocatable;
    module :FMS:KER286A0/ADAUS.LIB         relocatable;
    module GATELBA                         relocatable;
end subprogram;

<optional_subprogram_declaration> ::=
    subprogram ZONE_<i> optional;
    <zone_list>;
    { <module> };
    module :FMS:DACS86A0/ROOT.LIB          relocatable;
    module :FMS:DACS86A0/RTHelp286.LIB     relocatable;
    module :FMS:ADABIOA1/BINDER286.LIB     relocatable;
    module :FMS:ADARTSA1/BINDER286.LIB     relocatable;
    module :FMS:KER286A0/ADAUS.LIB         relocatable;
    module GATELBA                         relocatable;
end subprogram;

<repeated_list> ::= $ INCLUDE (REPEATED)
<zone_list>      ::= $ INCLUDE ( ZONE_<zone_identifier> )
<zone_identifier> ::= 0 | 1 | 2 | . . . | 15

<program_activation> ::=
    <activation_modality>
    <allocation_directive>
    <initial_process_declaration>
<activation_modality> ::= include <template_name>;
                       | invoke <template_name>;

<allocation_directive> ::= $ INCLUDE (SUBPROGRAMASSIGN);

```

"\$ INCLUDE" directives must be placed in column 1.

"INVOKE" activation specifications require a function declaration which includes their declarative (see [SCL]).

"Repeatable" subprogram contains the common utility code (for instance the libraries which allow Ada program to use Run Time Support services, the code which provides for compilation units processing, etc.). The i-nth optional subprogram includes the whole and only the

code that it has to execute on the i-nth processor.

In [APP B] section, MULTI\_ADA286 program SCL text is reported. The rules described here have been applied to this text.

### 3.4 Linker use in multi generation

In Ada programs multi generation, the use of automatic mode is recommended as it makes generation easier.

As in the case mono, a default graph to which 'ada'\_graph logic is associated, which exports a multi architecture [APP. B] is delivered.

If the user decides to adopt its own graph for generations, it must reassign 'ada'\_graph logic to this graph.

The templates of 'ada'\_graph that the user wants to use for multi generation are to be builded following these rules:

1. The name of the template must begin with the prefix MULTI\_ because the linking process adds always MULTI\_ to the template name specified in the invoking command.

For the use of /TEMPLATE qualifier in multi generations the same rules of case mono applies:

A) If /TEMPLATE is not specified in linker invocation command, the following rule is valid:

<template name> = MULTI\_<graph name>

(as Ada linker, for multi generations, which can be recognized by means of the use of /CLUSTERIZATION and /PROCESSOR\_ASSIGNMENT qualifiers, assigns as template name composed of the prefix MULTI\_ followed by the name of the graph.  
(see rule a) par. 1.1.2.1).

• Example

```
$ define ada286_graph MYGRA.GRA
$ Ada286/link/ofd=.../clu=.../proc=... MYPROC
```

means that the template which is to be used in MYGRA.GRA is:

<template name> = MULTI\_MYGRA

B) If /TEMPLATE is partially specified, the following rule is valid:

<template name> = MULTI\_<main unit>

Ada linker, for multi generations which can be recognized by means of the use of /CLUSTERIZATION and /PROCESSOR\_ASSIGNMENT qualifiers,

assigns to the template a name composed of the prefix MULTI followed by the name of the main unit (see rule b) par. 1.1.2.1).

• Example:

```
$ define ada286_graph MYGRA.GRA
$ ada286/link/ofd=../lib=../clus=../proc=../template MYPROC
```

means that the template which is to be used in MYGRA.GRA is

<template name> = MULTI\_MYPROC

C) If /TEMPLATE is totally specified

/TEMPLATE = <template>

the following rule is valid:

<template name> = MULTI\_<template>

Ada linker, for multi generations, which can be recognized by means of /CLUSTERIZATION and /PROCESSOR\_ASSIGNMENT qualifiers, assigns to the template a name composed of the prefix MULTI followed by the name of the template (see rule c) par 1.1.2.1).

• Example:

```
$ define ada286_graph MYGRA.GRA
$ ada286/link/ofd=../lib=../clu=../proc=../temp=MYTEMP MYPROC
```

means that the template which is to be used, in MYGRA.GRA graph is:

<template name> = MULTI\_MYTEMP

NOTE:

If linker invocation command is:

```
ada286/link/ofd=../lib=../clu=../proc=../temp=MULTI_MYTEMP MYPROC
```

then

<template name> = MULTI\_MULTI\_MYTEMP

MYGRA.GRA graph used in the examples above, is done as follows:

SCL text:

```
system MYGRA;
program template MULTI_MYGRA..
end program;

program template MULTI_MYPROC..
end program;
```

```

program template MULTI_MYTEMP..
end program;

program template MULTI_MULTI_MYTEMP..
end program;

```

...

```

end system;

```

- The use of CG\_ADAMAINPROGRAM as "initial procedure" name to be given in SCL is recommended.

- The use of ELABORATION (exported by the anonymous task) as name of the assembler module to be included in GENERAL subprogram is recommended.

2. The following logics must be defined before of the graph compilation, otherwise SCL286 Compiler fails.

The logics to be defined are:

- REPEATED must be defined on a file REPEATED.INC initially empty.
- ZONEi (i = 0..15) must be defined on a file ZONEi.INC initially empty
- SUBPROGRAMASSIGN must be defined on a file SUBPROGRAMASSIGN.INC initially empty

The ada linker will associate the \*.INC files to the oportunes .OBJ

3. The name of the "initial procedure" given in the example of [APP B] is CG\_ADAMAINPROGRAM which is an equ defined on CG?ADAMAINPROGRAM. CG?ADAMAINPROGRAM is the name of the public procedure of <main>\_ELAB.ASM which gives the elaboration order of an Ada program.

4. As name of the assembler module to be included in the GENERAL subprogram the user can use ELABORATION ([APP B]). This is a logic that the linker will provide to assign to the last <main>\_ELAB.ASM produced.

If there is the use of the qualifiers /SEARCHLIB and /INTERFACED in a multi generation there is the insertion of the specified interface modules (object files or libraries) only in the subprogram GENERAL of type repeatable.

It is impossible to insert interface units not ADA in types of subprogram not repeatable because the last cannot be inserted in the cluster specifications.

The names of the interface files specified with with the qualifiers above must follow SCL syntax, in fact they must be processed by SCL286 compiler.

### 3.4.1 Linker Use Example

Ada multiprocessor programs generation.

Given the following Ada text:

```
with Ada_RTS; use Ada_RTS;
package COMPUTER_0 is
  task PROCESSOR_0 is
    entry ARE_YOU_THERE(no : out computer_id);
  end PROCESSOR_0;
end;
package body COMPUTER_0 is
  task body PROCESSOR_0 is
    begin
      accept ARE_YOU_THERE(no : out computer_id) do
        no:= my_computer;
      end ARE_YOU_THERE;
    end PROCESSOR_0;
end COMPUTER_0;

with Ada_RTS; use Ada_RTS;
package COMPUTER_1 is
  task PROCESSOR_1 is
    entry ARE_YOU_THERE(no : out computer_id);
  end PROCESSOR_1;
end;
package body COMPUTER_1 is
  taskbody PROCESSOR_1 is
    begin
      accept ARE_YOU_THERE(no : out computer_id) do
        no:= my_computer;
      end ARE_YOU_THERE;
    end PROCESSOR_1;
end COMPUTER_1;

with Ada_RTS; use Ada_RTS;
package COMPUTER_2 is
  task PROCESSOR_2 is
    entry ARE_YOU_THERE(no : out computer_id);
  end PROCESSOR_2;
end;
package body COMPUTER_2 is
  task body PROCESSOR_2 is
    begin
      accept ARE_YOU_THERE(no : out computer_id) do
        no:= my_computer;
      end ARE_YOU_THERE;
    end PROCESSOR_2;
end COMPUTER_2;
```



```

with REPORT; use REPORT;
with Ada RTS; use Ada RTS;
with COMPUTER_0; use COMPUTER_0;
with COMPUTER_1; use COMPUTER_1;
with COMPUTER_2; use COMPUTER_2;
procedure FIRST is
  which_computer : computer_id;
begin
  text( "FIRST", "Processor migration occurs"&
    "during context processing");
  comment("PROCESSOR 0 is active");
  PROCESSOR_0.ARE_YOU_THERE(which_computer);
  if which_computer /= 0 then
    failed("PROCESSOR_0 is not on processor 0");
  end if;
  comment("PROCESSOR 1 is active");
  PROCESSOR_1.ARE_YOU_THERE(which_computer);
  if which_computer /= 1 then
    failed("PROCESSOR_1 is not on processor 1");
  end if;
  comment("PROCESSOR 2 is active");
  PROCESSOR_2.ARE_YOU_THERE(which_computer);
  if which_computer /= 2 then
    failed("PROCESSOR_2 is not on processor 2");
  end if;
  result;
end first;

```

Clusterization file:

```

FIRST
*
    FIRST

POLO_0
*
    COMPUTER_0

POLO_1
*
    COMPUTER_1

POLO_2
    COMPUTER_2

```

File of assignment to processors:

```

FIRST 0001
POLO_0 0001
POLO_1 0002
POLO_2 0004

```

#### 4 Ada Run Time Supports

Ada Run Time Supports for Ada applications execution are mainly located in Kernel Program of the Initial Program of an Alenia target computer. This means that the application needs very few codes to gain access to such run time supports.

This code is defined by the following set of OMF modules which are part of the cross compiler installation kit.

module	:FMS:ADARTSA1/RTSDATA.OBJ	relocatable;
module	:FMS:ADABIOA1/BIODATA.OBJ	relocatable;
...		
module	:FMS:ADABIOA1/BINDER286.LIB	relocatable;
module	:FMS:ADARTSA1/BINDER286.LIB	relocatable;
module	DACS86A0:RTHelp286.LIB	relocatable;

The first two modules define a data segment reserved to system data. It is accessed only by run time system during the program elaboration. The last three modules define all run time support services required by Ada program elaboration.

The user must assure that the two .OBJ modules and the three .LIB modules, respectively go first and follow all compiled Ada code (see [DPG]).

Accordance with the above requirements enforces first the correct relocation of run time system data in the reserved data segment. Secondly, it guarantees the visibility of the procedural supports of the run time system which is delivered in the .LIB modules.

#### 5 TASK CONFIGURATION

Any task declaration is configurable with respect to the priority used by Run Time System to elaborate task and with respect to dynamic storage size and allocation required by task elaboration. We briefly recall the way to give default values for following configuration items which must be supplied in each Ada program generation.

- 1) task priority;
- 2) task segment size and allocation;
- 3) task stack size;

Few examples guide the user in introducing this information into the program.

##### 5.1 Task Priority

For the current version of ADARTS and for the old ones the following rules applied.

1. Default priority of the anonymous task id fixed at SCL time with the priority directive.

initial process on 0 priority 2;

It can be modified using the pragma priority with values from 0 to 7. Note that priority'FIRST is equal to 1 (for the current version).

2. All other tasks start with a priority equal to the SCL value specified for the main program, this priority can be changed by the use of pragma PRIORITY. Inner task inherits the priority of parent task.

## 5.2 Task Segment Size

A stack segment is reserved for each library task and for each main program.

For each task is reserved a portion of stack (Stack Branch) in the stack segment; the branch stack for an inner task (that is a task whose implicit or explicit type is contained in an Ada frame - Ada Reference Manual par 11.2) is allocated in the same stack segment of the task which has activated it.

The size of the stack segment of the main can be specified in the following ways:

- STACKSEGMENT SIZE = N

(SCL statement; N hex value): it is the size in bytes of the stack segment created for the Main Task.

Its value is specified in the Program Template of the Ada program. The linker qualifier /MP\_SEGMENT\_SIZE = N has no effect.

scl text:

```
/* scl declarative part.  
*/
```

```
initial procedure CG_AdaMainProgram  
stacksegment nodal size = OFF00H;
```

The size of the stack segment for a library task is specified by:

- /LT\_SEGMENT\_SIZE = N

(linker qualifier): specifies default segment size for all library tasks to be N words (decimal value).

If this value is not specified the default is the size expressed in SCL for the main.

- pragma LT\_SEGMENT\_SIZE(T, N)

(part of Ada source text): specifies segment size for library task T to be N words (decimal value) if it is a Library Task.

- Function SET\_CHILD\_SEGMENT\_SIZE exported from the package ADA\_RTS ([APX A]);  
the size is expressed in word.

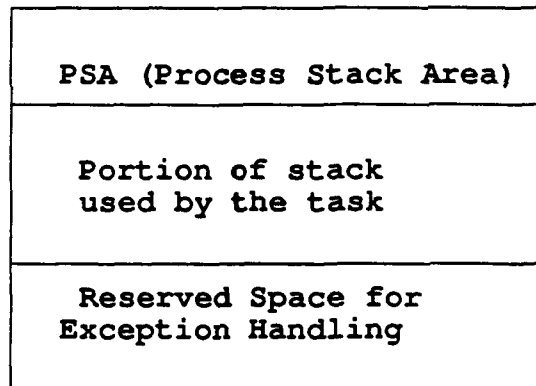
The size of the stack segment includes always the Page Map that is the two pages reserved on the stack for the Run Time Support of Ada.

Furthermore, the stack segment size specified and the stack segment size actually used aren't the same, because the ADARTS provides to add some words for page alignment.

### 5.3 Task Stack Size

For each task is reserved a portion (Task Branch) in the stack segment allocated for it.

A branch has the following structure:



The Process Stack Area is an area where are stored some task informations; the size of this area is 12H storage units for a library task, while is 10H storage units for an inner task.

Under the PSA there is the part of the stack used by the task to allocate the various blocks during its execution.

There is also a zone reserved for the exception handling and its size is 50 storage units.

If the Main Task or a Library Task have no inner tasks (i.e. tasks that are not library tasks), then the Branch Size (page\_aligned) can be up to the Stack Segment size minus 512; otherwise the sum of the Branch Size (page\_aligned) of all the inner tasks depending from the main or a Library Task plus 512 must be less than the Segment Size.

The Stack Branch Size of a task can be specified in the following way:

- /MP\_STACK\_SIZE = N

(linker qualifier): specifies stack size for the main program to be N words; it is the default value for /LT\_STACK\_SIZE.

- /LT\_STACK\_SIZE = N

(linker qualifier): specifies default stack size for library tasks to be N words.

- /TASK\_STORAGE\_SIZE = N

(linker qualifier): specifies default stack size for all non-library tasks stacks to be N words.

- for T'STORAGE\_SIZE use N

(part of Ada source text): specifies that the size of the stack for task type T should be N words.

The stack size specified and the stack size actually used aren't the same, because the ADARTS provides to add some words for page alignment

#### 5.4 Example

The following example illustrates how to use the previous parameters in order to set segment size and stack size.

```
package library_tasks is
  task type t1;
  pragma lt_segment_size(t1,1000);
  for t1'storage_size use 550;
  ttl:t1;
  task t2;
  -- segment size = value of /LT_SEGMENT_SIZE
  -- stack size   = value of /LT_STACK_SIZE
end library_tasks;
package body library_tasks is
  task body t1 is
    ...
  end t1;
  task body t2 is
    task t21;
    -- stack size = value of /TASK_STORAGE_SIZE
    task t21 is
      ...
    end t21;
  end t2;
end library_tasks;

with library_tasks;
proceiure main is
  task tml;
```

```

        -- stack size = value of /TASK_STORAGE_SIZE
task body tml is
begin
    null;
end tml;
    -- stack size of main = value of /MP_STACK_SIZE
begin
    null;
end main;

```

SCL text:

```

program template main
.
.
initial procedure main
    stacksegment nodal size = 0FF00H;
end program;

```

Suppose that the linker command has the following qualifiers:

```

/MP_STACK_SIZE      = 7000
/LT_SEGMENT_SIZE    = 8000
/LT_STACK_SIZE      = 600
/TASK_STORAGE_SIZE  = 900

```

The program given in the example contains a main task, two library task (TT1, T2) and two inner task (T21, TM1). Here there is the allocation of three stack segments:

- one stack segment with size = 0FF00H bytes for the main task (by STACKSEGMENT SIZE = 0FF00H)
- one stack segment with size = 1000 word for the library task TT1 (by PRAGMA LT\_SEGMENT\_SIZE)
- one stack segment with size = 8000 word for the library task T2 (by /LT\_SEGMENT\_SIZE)

Moreover there will be the allocation of the following branches inside the stack segment above mentioned:

- one branch stack for the main task with size = 7000 word (by /MP\_STACK\_SIZE)
- one branch stack for the library task TT1 with size = 550 word (by length clause)
- one branch stack for the library task T2 with size = 600 word (by /LT\_STACK\_SIZE)
- one branch stack for the inner task T21 with size = 900 word (by

/TASK\_STORAGE\_SIZE) in the stack segment of T2

- one branch stack for the inner task TM1 with size = 900 word (by /TASK\_STORAGE\_SIZE) in the stack segment of the main

## 6 Objects Dynamic Allocation

### 6.1 System Heap

System Heap is a segment in which collections without associated length clauses are allocated.

- Example:

```
type ARR1 is array (1..10) of integer;
type ARR2 is array (1..15) of integer;
type PUNT1 is access ARR1;
type PUNT2 is access ARR2;
.
var1 : punt1;
var2 : punt2;
```

Objects of ARR1, ARR2 type, created with NEW allocator, form a collection in which no length clause has been fixed, consequently they are allocated in System Heap.

System Heap segments are created as nodal or local according to indications in SCL concerning data segments allocation.

If data segments allocation has been fixed in nodal memory, System Heap segments will be nodal, too.

Similarly, if data segments allocation is in local memory, System Heap segments will be local.

### 6.2 Heap Segment

Collections with applied length clause are allocated in Heap Segments.

- Example:

```
type ARR1 is array (1..10) of integer;
type ACC1 is access ARR1;
for ACC1SIZE use 80;
VAR1 : ACC1;
```

The same rules valid for System Heap apply to Heap Segments allocations in nodal or local memory.

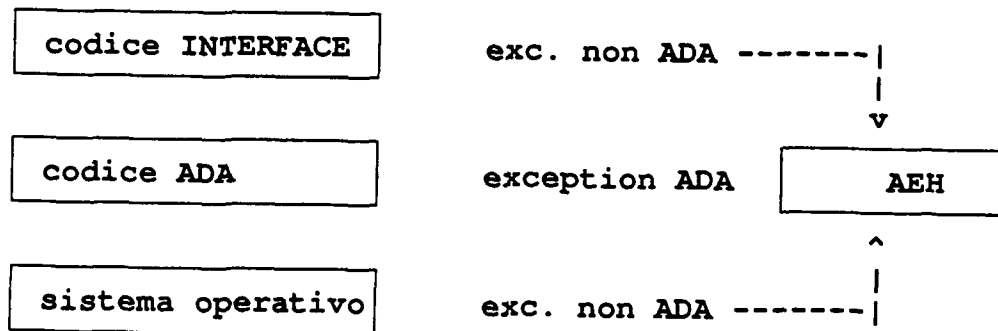
7

## Exceptions handling

This subsection describes process Exception Handling (AEH) and function Exception Handling supplied by Run Time Support for the execution of ADA programs.

AEH is the process Exception handling associated to any process which implements an ADA task.

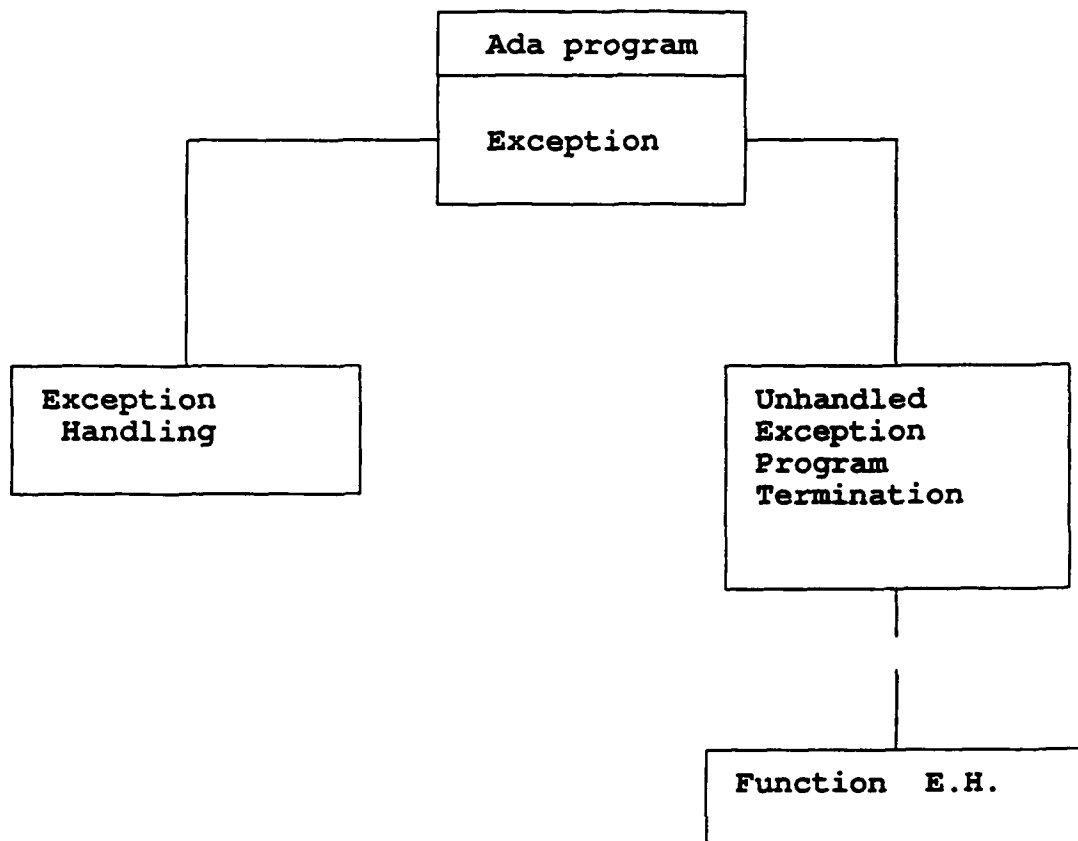
It is activated whenever there is an exception, whether it is Ada or is not Ada.



- fig. 1 -

Function Exception Handling is activated by RTS when main program processing concludes because of unhandled exceptions





- fig 2 -

### 7.1 Process Exception Handling

Ada Exception Handling (AEH) is part of Run Time Support (RTS), whose task is to perform exceptions handling policies according to description in [LRM] chapter 11.

AEH handling policies vary according to the following factors:

- frame typology (see 11.2 - [LRM])
  - block statement
  - body of subprogram, package, task, generic unit

- statement typology
  - declarative
  - executive
- presence or absence of 'exception handler'

AEH must search a handler for the raised exception and consequently perform the following actions:

- if there is a handler in the damaged frame:
  - transmit control to it
- if there is no handler in the damaged frame, look after:
  - propagating exception until handler is detected, without overcoming task or main program level
  - terminating the program in execution, when propagation is arrived at main level

## 7.2 Function Exception Handling

Function Exception Handling is activated when a specific exception, being not handled, propagates until the most external program level causing, as a consequence, the termination of the program itself.

Each function possesses a default function exception handling.

Anyway, implementation makes a function and non-default Exception Handling available, which executes main history layout starting from the damaged block.

It is actually an RTS optional component which the user can profit by only if it has been explicitly activated.

Activation specification and function EH layout structure supplied by the implementation are described in section 6.6 .

## 7.3 Interaction specification with process EH

This chapter describes AEH different behaviour depending on execution code, that is Ada code or non Ada code performed by Ada task through subroutine calls to which Pragma INTERFACE has been applied

Exceptions can be raised during Ada task execution:

- caused by Ada code execution
- caused by non Ada code execution

The former case includes all those anomalous circumstances described in terms of the 5 predefined exceptions as well as those anomalous



## Case 2 - exceptions in non Ada code

```
MAIN:do;
RAISEEXCEPTION : procedure (p1,p2,p3,p4) external
  declare (p1,p2,p3,p4) word;
end RAISEEXCEPTION;
FOREIGN : procedure public;
...
  CALL raiseexception(0E081H,0,0,0);
  /* 0E081H is raised here */
end FOREIGN;
end MAIN;
```

```
procedure PROVA2 is
  procedure FOREIGN;
  pragma interface (PLM_ACF, FOREIGN);
begin
  FOREIGN;                                - 0E081H is raised here
exception
  when STORAGE_ERROR                     => ... ;
  when others                             => ... ;
end PROVA2;
```

- fig.4 -

The handling of exceptions raised during non Ada code execution is performed by AEH through exception propagation into Ada code and through a default mechanism which translates each non Ada exception into one of the 5 predefined Ada exceptions.

In addition, the programmer may also define his own transposition mechanism to translate non Ada exceptions into predefined and not Ada exceptions.

In this case, the programmer himself must lead AEH to use such non-default associations. The way to implement this will be shown later on in this paragraph.

Transposition mechanism supplied by AEH, what will be later called default, performs the following associations:

A030H	storage_error
C031H	program_error
C032H	program_error
C034H	program_error
C035H	program_error
C03BH	program_error
A03CH	storage_error
A03DH	storage_error
A03EH	program_error
A042H	program_error
C052H	program_error
C06EH	constraint_error
E06FH	program_error
E073H	program_error
C075H	numeric_error
C076H	numeric_error
E079H	program_error
C080H	program_error
E081H	storage_error
C0F0H	numeric_error
E0F1H	program_error
A410H	storage_error
A411H	storage_error
A412H	storage_error
A414H	storage_error
C421H	program_error
C422H	program_error
C423H	program_error
C427H	program_error
C429H	program_error
C42EH	program_error

- fig. 5 - Default association table

Note that all non Ada exceptions codes present in default table are translated by AEH into FOREIGN\_EXCEPTION Ada exception defined in package SYSTEM.

Therefore, the previous example in fig. 4 can be updated following use in the given table, as shown :

Case 2:

```
MAIN:do;
RAISEEXCEPTION : procedure (p1,p2,p3,p4) external;
declare (p1,p2,p3,p4) word;
end RAISEEXCEPTION;
FOREIGN:procedure public;
...
CALL raiseexception(0E081H,0,0,0);
/* 0E081H is raised here */
end FOREIGN;
end MAIN;

procedure PROVA2 is
procedure FOREIGN;
pragma interface (PLM_NOACF,FOREIGN);
begin
FOREIGN;                                - STORAGE_ERROR is reraised here
exception
when STORAGE_ERROR                      =>...; - STORAGE_ERROR is handled here
when others                             =>...;
end PROVA2;
```

- fig . 6 -

Whenever the user need to define different associations from default ones, than he must write a table the way is shown in figure 7 :

Ada compilation unit
Table size
non Ada exception codes
Ada exception codes

- fig.7 - Association table

'Ada compilation unit' field must contain the number (taken in ADA sublibrary ) of compilation unit in which has been declared.

The number of compilation unit can be found through 'Program Library Utilities' (see chap. 4 [User's Guide]).

'Table Size' is the number of exceptions codes listed in one of the following subtables.

In 'Non Ada exception codes' field non Ada exception codes must be listed , that the user wants to translate based on his associations.

In 'Ada exception codes' field Ada exceptions codes are listed (predefined and not), chosen by the user to translate the errors list given in the previous field.

Each element in the assosiation table must occupy a 16 bits space.

'Non Ada exception code' and 'Ada exception codes' must clearly contain the same codes number (one to one association between the two subtables entries).

Figure 8 reports the example of association table supplied by the user:

0
3
100
E081
C080
1
3
4

fig 8

Note:

Each exception is defined through a double word (definition is given in package STANDARD)

one word containing the compilation unit defining it

one word containing progressive declaration order

From table in figure 8 we desumw that:

1) The compilation unit is that of package STANDARD ( 0 is it number in root sublibrary) which contains the definition of the 5 predefined Ada exceptions

2) Entry number of each subtable is 3

3) Implemented associations are:

```

0100H --> 0:1    or CONSTRAINT_ERROR
0E081H --> 0:3    or PROGRAM_ERROR
0C080H --> 0:4    or STORAGE_ERROR

```

The user must lead AEH to use its own non\_default association table (not default one) and this occurs when you give significant values to RaiseException procedures parameters and precisely the following:

- The value of P1 parameter must be such that, after performing a logic AND with a "1700" value it preovides still "1700" result (P1 and 1700 = 1700).

- "P2" and "P3" parameters must contain the pointer to association table (P2 = selector and P3 = offset).



- "P4" parameter must be exception code as is listed in "Non Ada exception codes" subtable.

Let's analyze now the following example:

```

package PACK is
  procedure FOREIGN;
  CONSTRAINT_ERROR :exception;
  OLD_EXCEPTION    :exception;
  MEW_EXCEPTION    :exception;
private
  pragma INTERFACE (PLM_ACF, FOREIGN);
end PACK;

WITH pack;
procedure PROVA is
begin
  PACK.FOREIGN;
exception
  when PACK.CONSTRAINT_ERROR => null;
  when OLD_EXCEPTION        => null;
  when others
end PROVA;

```

Suppose that:

- 8193 is PACK compilation unit number
- association table supplied by the user is structured as follows:

8193
4
200 0C06E 0C0F0 0A06D
1 2 2 3

- fig.10 -

The user chooses the following associations:

0200H	-->	8193:1	or	CONSTRAINT_ERROR
0C06EH	-->	8193:2	or	OLD_EXCEPTION
0C0F0H	-->	8193:2	or	OLD_EXCEPTION
0A06DH	-->	8193:3	or	NEW_EXCEPTION

In order to use his own association table, the user must:

- declare the table
- perform calls to RaiseException according to the rules given before

Here is an example of use:

```
MAIN:do;
RAISEEXCEPTION:procedure (P1,P2,P3,P4) external;
declare (P1,P2,P3,P4) word;
end RAISEEXCEPTION;
FOREIGN: procedure public;
declare TABLE (*) word data (      8193H,
                                     4,
                                     200H,
                                     0C06EH,
                                     0C0F0H,
                                     0A06DH,
                                     1,
                                     2,
                                     2,
                                     3 );

declare PTR pointer;
declare (MYOFF,MYSEL) word at (@PTR);
declare RESULT word;

PTR = @TABLE;
/* <call to Kernel procedures> */
if RESULT <> 0 then
call RAISEEXCEPTION(1701H,MYSEL,MYOFF,200H);
/* <call to Kernel procedures> */
if RESULT <> 0 then
call RAISEEXCEPTION(1701H,MYSEL,MYOFF,0C06EH);
end FOREIGN;
end MAIN;
```

- fig.11 -

- NOTE: CONSTRAINT\_ERROR exception declared in PACK package is different from CONSTRAINT\_ERROR declared in STANDARD package.  
While the former is identified by (8193:1), the latter is identified by (0:1).

In some cases process AEH behaviour is not dictated by language rules, but depends on implementation. Particular cases are analysed below:

- overflow situations in real or fixed type operations are not declared by `NUMERIC_ERROR`, but the user can detect them using `MACHINE_OVERFLOW`s attribute (see C.1.3 Users Guide, see. 4.5.7 - Reference Manual).
- memory violation is declared as `PROGRAM_ERROR`. stack
- overflow and underflow cases are declared as `STORAGE_ERROR`.

#### 7.4 Use of Exception\_Code

ADA RTS package, present in Ada Program Library, exports a function called `Exception_Code`, which returns in an `UNSIGNEDWORD` type variable (defined in package `SYSTEM`) the code of the last non Ada exception raised during a program.

If no non Ada exception has been raised, it returns value 1700H.

This function is particularly useful in case the association performed by AEH or chosen by the programmer is such that `FOREIGN_EXCEPTION` is notified as non Ada exception raised during the execution of a program.

An example of use is given below:

```
with SYSTEM;use SYSTEM;
with ADA_RTS;use ADA_RTS;
with TEXT_IO;use TEXT_IO;
procedure P_PROC is
  procedure FOREIGN;
  pragma interface (PLM86,FOREIGN);
  package PUT_UNSW is new INTEGER_IO(UNSIGNEDWORD);
  use PUT_UNSW;
  MARA_EXC : UNSIGNEDWORD := 0;
begin
  FOREIGN; - raises STORAGE_ERROR;
exception
  when STORAGE_ERROR => MARA_EXC := EXCEPTION_CODE;
    PUT_LINE ( The original code &
               of non Ada exception is: );
    PUT(MARA_EXC,BASE=>16);
    NEW_LINE;
  when others
    => PUT_LINE( other exception );
    NEW_LINE;
end P_PROC;
```

## 7.5 Printout Of Exception Spelling

The package ADA\_RTS contains two function:

- RTS\_GetExceptionId
- RTS\_GetExceptionSpelling

which are necessary to obtain the complete name of the last exception occurred.

In fact from the release 4.6, ADA compiler allows the retrieval of full spelling for raised exceptions.

The linker extracts all exception spellings from all ingoing units and places them in the elaboration code module from which they may be identified. So there is the use of two routines: one to retrieve the last raised exception (RTS\_GetExceptionId) and one to translate an exception identifier into a string (RTS\_GetexceptionSpelling).

Here is an example of the use of these procedures in a program where there is the declaration of some exceptions:

Example:

```
with Text_IO; use Text_IO;
with System; use System;
with Ada_rts; use Ada_rts;
procedure excspell is

    temp_outside_limit:exception;
    fire, break_in:exception;
    stack_overflow, stack_underflow:exception;

begin

    raise fire;

    exception

    when fire =>
        put (RTS_GetExceptionSpelling (RTS_GetExceptionId));

end excspell;
```

## 7.6 Use of pragma INTERFACE

Pragma INTERFACE form is the following (see C.2.4 - [User Guide]):

pragma INTERFACE (language\_name, subprogram\_name).

<language\_name> types allowed by the implementation are defined in package SYSTEM (see. F.3 - Users Guide).

<Language\_name> can be followed by the following annotation:

\_ACF or \_NOACF.

\_ACF annotation notify to AEH that you have passed from Ada code to non Ada code, therefore exceptions handling policy must follow the rules valid for this case and expressed previously in this section.

If <language\_name> is of \_NOACFS type, AEH cannot individuate Ada code from non Ada code and its exceptions handling politic follows given rules for handling of exceptions from Ada code, ignoring completly the fact that it works in non Ada context.

This situation does not preserve the user from incorrect exceptions handling.

PLM86 and PLM\_ACF annotations are equivalent, in fact both ensure exceptions handling.

Similarly for C86 and C\_ACF.

On the contrary, for ASM86<language\_name> the rule is the following:

ASM86 is equivalent to ASM\_NOACF, while ASM\_ACF use ensures the handling of exceptions raised in non Ada code.

## 7.7 Use of Trace\_info Procedure

From the version 1.5 of ADARTS and from the version 4.5.4.1 of ADA compiler it's possible to use a procedure which gives some informations related to the address and the identifier of an exception raised in an ADA program.

The name of this procedure is : Trace\_Info .

The specification of this procedure is contained inside the package ADA\_RTS, the body is written in ASM286.

The procedure Trace\_Info has a parameter (in out) of a type record declared in the package ADA\_RTS too.

The spacification of the procedure is :

```
procedure TRACE_INFO(PSA : in out PSA_REC);  
pragma interface (ASM286, TRACE_INFO);
```

The declaration of the record type is:

```
type PSA_REC is  
  record  
    Unit_no           : system.unsignedword;  
    Exception_id      : system.unsignedword;  
    Mara_code         : system.unsignedword;  
    Exception_offset  : system.unsignedword;  
    Exception_selector : system.segmentid;  
  end record;
```

[NOTE - the type UnsignedWord is defined in the package SYSTEM.]

The fields UNIT\_NO and EXCEPTION\_ID of PSA\_REC give some informations about the exception identifier.

The field MARA\_CODE give the original code of the exception.

The fields EXCEPTION\_SELECTOR and EXCEPTION\_OFFSET give the address of the exception.

The TRACE\_INFO procedure must be called inside an Ada Exception Handler defined in the same level or in a more external level respect to the block which has raised the exception.

The following procedure give an example about the use of the Trace\_Info.

```
with TEXT_IO; use TEXT_IO;
with SYSTEM; use SYSTEM;
with ADA_RTS; use ADA_RTS;

procedure main is
  psa: psa_rec;
  package my_uns is new integer_io(unsignedword);
  use my_uns;
begin
  raise storage_error;
exception
  when storage_error => ada_rts.trace_info(psa);
                        put_line("the exception address is:");
                        my_uns.put(psa.exception_selector,base =>16);
                        my_uns.put(psa.exception_offset,base =>16);
                        new_line;
                        put_line("the original exception is:");
                        my_uns.put(psa.mara_code,base =>16);
                        new_line;
                        put_line("the exception identifier is:");
                        my_uns.put(psa.unit_no,base =>16);
                        my_uns.put(psa.exception_id,base =>16);
end main;
```

## 7.8 Interaction specification with function EH

This paragraph shows behavior of optional function EH supplied by the implementation and the ways to set it.

It is called optional because it works only after it has been explicitly set, otherwise default or previously set one is inherited (if setting has occurred).

Function EH provided by RTS executes a main program evolution tracing, terminated because of

unhandled exception, from the exception point up to its deactivation.

Tracing shows the following information:

- identifier of the raised exception
- identifier of the damaged block which can be a block statement or a subprogram.

As regards the subprogram, the subprogram starting address is supplied, too.

The last information is repeated for all activated blocks until main program is attained.

The following example reports an Ada program tracing which does not include the handling of possibly raised exceptions:

```
procedure APPLICATION is
procedure INTERNAL is
begin
raise PROGRAM_ERROR; - not handled
end INTERNAL;
begin
begin                - inner block
INTERNAL;             - subprogram call
end;
end APPLICATION;
```

- fig. 13 -

Tracing is:

#### START OF TRACING

Unit number and exception identifier: 0000:0003

Unhandled exception raised at: 0067:0039

Trace back follows:

BP value is: FDB6

Subprogram entry point is: 0067:0028

BP value is: FDC8

Inner block

BP value is: FDD8

Subprogram entry point is: 0064:0010

BP value is: FDDE

Subprogram entry point is: 007C:0008

BP value is: FFFF

END OF TRACING

- fig. 14 -

Unit number and exception identifier is the exception identifier.

Unhandled exception raised at provides the exception original address.

BP value is is BP register value (BP detects frame univocally).  
Subprogram entry point is is the subprogram starting address (if the  
frame is a block statement, inner block nformation will be  
visualized).

Optional function EH is visible to the user as an Ada procedure called  
Unhanded\_Exception\_Tracer and is exported from ADA\_RTS package  
present in Program Library.

Therefore, setting function EH means simply to perform a call to  
Unhanded\_Exception\_Tracer procedure.

It is important to consider that such function EH works from call  
point to Unhanded\_Exception\_Tracer procedure and that therefore, only  
from that moment onwards is it possible to get the tracing of  
applications which are performed later on.

There are various ways to set such function EH.

You can either generate a program in LTL format to be performed before  
any other application program, or generate an IPL program having as  
Initial Program the procedure which performs call to  
Unhanded\_Exception\_Tracer.

We report the simpler example, the generation of a unique LTL for the  
following program:

```
with ADA_RTS; use ADA_RTS;
procedure TRACE is
begin
  UNHANDLED_EXCEPTION_TRACER;
end TRACE;
```

- fig.15 -

The execution of TRACE program LTL provides to set function EH  
supplied by the implementation.

From now on all performed Ada application programs will be traced as  
shown in figure 14.



It is possible to imagine more complex situations than the above one in which call to `Unhanded_Exception_Tracer` procedure can be inserted, but the programmers greater effort consists in detecting its application point in which tracing can be activated, because it is only from that moment onwards that it can be implemented.

## 8 Concurrency tracing

Support to the execution of Ada codified programs -`ADA_RTS`- provides a tracing performance of points relevant to program tasking activity.

From now on, we will refer generically to these points calling them synchronization points of global tasking programmed in the application.

This service is performed with the output - on `TRACER_DEVICE` path - of text lines whose meaning and format is described in `trace meaning` and `trace structure` sections.

`TRACER_DEVICE` is defined in (SCL) product static configuration phase.

The availability of this path is necessary for this service to be executed.

This is an optional service. Whether it is included or not in the Initial Program of a MARA node, this service is ruled by means of `ADARTS_TRACING` further parameter of the above configuration.

### 8.1 Trace lines structure

A trace line is structured as follows:

```
<line> ::= <action> <task> in <entry> at <time> [ for <delay> ]
```

```
<action> ::=      Initialized program :
| Identified
|   Awaked
|   Slept
|   Timed slept
|   Deleting
|   Creating
| Terminated program :
```

```
<task>      ::=      <word>
```

```
<word>      ::=      <hex><hex><hex><hex>
```

```
<hex>       ::=      0 | 1 | ... | E | F
```

```
<entry>     ::=      <identifier>:<address>
```

```

<identifier> ::= E_INIT | E_TERM | E_CRET | E_ACTT | E_ACTD
| E_LVBL | E_LBKT | E_LVMB | E_COMP | E_DELY
| E_CEUN | E_CETI | E_SLCT | E_ACPT | E_SYNC
| E_RVCO | E_RVFA | E_ABRT | E_STSZ | E_TRMD
| E_CABL | E_ECNT | E_TIDN | E_COMP | E_CHLC
| E_SGSZ | E_CHSS | E_ECOD

```

```

<address> ::= <word>

```

```

<time> ::= <word><word>

```

```

<delay> ::= <word><word>

```

Example A:

Initialized program : 000C

Identified 000C in E\_COMP:072E at 005B11DE

Identified 000C in E\_TERM:1886 at 005B1242

Terminated program : 000C

## 8.2 Trace lines meaning

The elaboration of an Ada text programming one or more of the definitions described in chapter 9 of language manual, determines the univocal identification of points in the text in which actions described in that chapter must occur.

These actions are performed by Run Time System (RTS) from appropriate procedures directly called by the compiled code, i.e. produced by Ada compiler.

These procedures are directly performed by the task processing Ada text at issue, and trace lines provide

<action><word> identification of that task.

When the tracing mechanism is present, these procedures trace the run of the task in process outputting the above lines in order to indicate that the corresponding action is occurring.

Relevant events which are traced can be roughly classified in the following groups:

- Task creation and activation
- Communication between tasks
- Task dependences and termination
- Time management
- Task abnormal termination
- Tasks attributes

Some examples, pointing out the correspondence between a given Ada text and a possible tracing which can be obtained through its elaboration, are reported below.

The reader should carefully think about the term possible of the previous statement, taking into account that tasking means parallel executions .

For compactness reasons, Ada text can be replaced with the corresponding syntactic non\_terminator , as shown in language manual.

The Ada text examples are from the language manual.

The mark => indicates the beginning of the expected layout after the elaboration of the previous text.

### 8.3 Example of Task declaration

```
task type keyboard_driver is
  entry read(c : out character);
  entry write(c : in character);
end;
teletype : keyboard_driver;
=>
Identified <task> in E_CRET:<word> at <time>
```

### 8.4 Example of Task activation

```
A)
task resource is
  entry seize;
  entry release;
end;

procedure p is
  a,b : resource;
  c : resource;
begin
  - A, b, c tasks are concurrently activated before the
  - first instruction
=>
Identified <task> in E_ACTT: <word> at <time>
```

B)

```
type keyboard is access keyboard_driver;
terminal : keyboard := new keyboard_driver;

=>
Identified <task> in E_CRET:<word> at <time>
Identified <task> in E_ACTT:<word> at <time>
```

C)

```
task body protected_array is
    table : array(index) of item := (index => null_item);
begin
=>
Identified <task> in E_ACTD:<word> at <time>
```

## 9 TASKING AND TIME HANDLING

### 9.1 REAL\_TIME\_CLOCK Package

Provide for direct access to kernel Real Time Clock GetTime and SetTime procedural services.

The Type TIME counts the milliseconds in a day.

TIME\_IO package can be used to exchange the values of type TIME objects with STANDARD\_INPUT and STANDARD\_OUTPUT. In case several tasks need to perform concurrent GET or PUT requests, the provision of a monitor task for TIME\_IO package is recommended.

Example - REAL\_TIME\_CLOCK

```
With REAL_TIME_CLOCK; use REAL_TIME_CLOCK;
with TEXT_IO; use TEXT_IO;
procedure CLOCK is
    I : INTEGER;
    use TIME_IO;
begin
    for I in 1..10 loop
        PUT (PUTTIME);
        NEW_LINE;
    end loop;
end;
```

## 10 ADABIO PRODUCT

It is the IO support for the execution of Ada programs generated by Ada-DDC compiler 4.3 and following versions.

This version is compatible with:

firmware	v.4.0
genentry	v.4.0
scl26	v.4.0
genrp	v.4.0
genip	v.4.0

abs286	v.3.0
ker286	v.4.0
io286	v.4.0
adarts	v.1.0

and following versions.

In order to generate correctly this product, the following logic names must be given to the directories described below:

ADABIOA0 containing ADABIO sources  
ADABIOA1 containing ADABIO object modules  
  
INTELA0 containing INTEL factory modules  
  
ADARTSA0 containing Ada Run Time Support modules  
(ADARTS)  
  
KER286A0 containing Kernel modules

For additional information, refer to ADARTS PRODUCT.

## 11 Input Output Packages

### 11.1 Ada File Objects and Mara External Files

Note the difference between:

- Object file: entity create by ADA program for I/O operation that last until the program termination

- External file: ADA environment external entity that last independently of the program and manage data to exchange

An Ada task can input or output Ada objects values from or towards Mara external files only after that these are connected to adequate Ada file objects declared in the program.

This connection can be operated with one of CREATE and OPEN primitives defined by i/o packages provided by the implementation. Each CREATE creates a new Mara file, and each OPEN opens a pre-existing file.

Given Mara external file bytes arrangement, the binary notation of values output towards a Mara file is operated in terms of bytes.

At present, no other Mara external file organization is used by the implementation for Ada objects values representation output towards a Mara file.

In the three following sections, implementation choices, concerning the representation of Ada objects values stored in Mara files with the three Ada input output standard models, are detailed.

These models are specified in SEQUENTIALIO, DIRECTIO, and TEXTIO

packages (see [LRM 83] chapter 14).

### 11.1.1 Ada Sequential Files

An Ada sequential file is a set of single type objects values which are accessible only by read operations according to a strict sequence in consecutive file positions. A previous write operation allows the creation of sequential files. These two modes are exclusive and cannot co-exist at the same time. RESET operation permits to modify file access by activating the transfers from the beginning of the file in the new mode. Sequential files OPEN activates the transfers from the beginning of the file.

This means that if it is a write open, the file previous contents are lost. RESET operation in OUTFILE mode acts as a new output open from the beginning of the file.

Therefore, file previous contents are lost. Reading sequential files accessed through a CREATE request is considered as wrong, and causes USE\_ERROR exception raising.

The number of Mara file bytes involved in a value recording is determined by associated types binary notation length.

Given the existence of composite types - whose binary notation has a width which is not statically known -, these types sequential files record these values on a number of bytes dimensioned on the maximum occupancy.

Example:

```
type index is range 1..10;
type an_array is integer array(index range <>);
package array_io is new sequential_io(an_array);
```

Maximum width value for an AN\_ARRAY object corresponds to 10 integers. An integer is represented by 16 bits. Thus, 160 bits are the width used to represent this value. Therefore, Mara files created through ARRAYIO package will record AN\_ARRAY types objects values on 20 bytes.

```
a,b    :   an_array      :=           (1,2,3);
c,d    :   an_array      :=           (1,2,3,4,5,6,7,8,9,0);
array_io.write(array_io.some_file,a);
array_io.write(array_io.some_file,c);
array_io.write(array_io.some_file,b);
array_io.write(array_io.some_file,d);
```

At the end of this sequence, 80 bytes of Mara file associated to ARRAY\_IO.SOME\_FILE will be occupied: the first 6 are significant, 14 are not significant, then 20 plus 6 are significant, 14 non significant, and finally 20 are significant.

### 11.1.2 Ada Direct Files

An Ada direct file is a set of values of single type objects which are accessible in reading or in writing in a file position selected by an index in the interval 1 ..DIRECT\_IO.POSITIVE\_COUNT'LAST.

This index control is explicitly operable through appropriate subprograms.

The index of the first element is 1, the last element one is file length. Conventionally, 0 length files are empty.

When opened, a direct file places the current index at 1.

Direct files can be extended by placing the index beyond file current size.

Unlike sequential files, direct files can be opened in INOUT mode, and this allows to update the file without the need to close or activate RESET operations.

### 11.1.3 Ada Text Files

An Ada text file is a set of pages which are a set of lines that, in turn, are a set of characters. A text file characters, lines, and pages can be selected by indexes ranging in the interval which goes from 1 to an integer value which can be defined by means of SET\_LINE\_LENGTH and SET\_PAGE\_LENGTH subprograms.

The lines in a page, and the pages in a text file cannot exceed the value TEXT\_IO.POSITIVE\_COUNT'LAST.

Ada TEXTIO includes the UNBOUNDED constant in order to handle text file lines and pages with an unbounded number of characters per line, and of lines per page.

Lines and pages are respectively separated by line terminators and page terminators which are represented by control characters sequences.

As to line terminator, the user can choose between <CR> and <CR,LF> ASCII characters sequences, the one which is more appropriate to the file at issue. This selection can be performed through the FORM string when the file is opened or created (see [BIO]).

Page terminator is codified with the <FF> ASCII character.

Text files termination is identified because of the absence of additional characters to be read. This means that:

- Read operations beyond the end of text files recorded on memory devices (disks, tapes, etc.) raise ENDERROR exception.
- Read operations addressed to communication lines - whose drivers do

not generate any signaling, for the communication session closing - stop the application.

#### 11.1.4 File Ada Objects Accessibility Criteria

It is necessary to observe that the piece of information output towards a Mara file created by an Ada program is correctly interpreted when read only if the following conditions are valid:

- a) Use the same file model as the one adopted for writing;
- b) As to generic models, instantiate with the same types as the ones used for writing;
- c) For unconstrained writing types, readings must operate regularly with the same constraints .

Exceptions to rule b) are possible if SEQUENTIAL\_IO and DIRECT\_IO are used. In this case, the particular recording choice allows a direct, or sequential access to single files positions which are respectively sequential, or direct.

Obviously, these rules can be ignored when there is a total knowledge of types representation. In this case, however, a reading operated in terms of bytes, even if it is onerous, is realizable. It could be the case of the check of files produced by non Ada programs, or by Ada programs generated by language implementations which are different from the one used for the reader programming.

#### 11.1.5 External Files Name - [NAME]

NAME parameter defined by any CREATE or OPEN primitives has to report a valid MARA path name.

Names indicating non-existent, or somehow illegal path servers cause an appropriate exception raising.

Ada Temporary Files (see [LRM] ch. 14.2.1) associated to name parameter null string, are associated to MARA File System work files. As a consequence, they are codified with the :FMS:\* string.

USE\_ERROR exception is raised by NAME function invoking for an Ada temporary file .

#### 11.1.6 Operating Modes On External Files - [FORM]

FORM parameter in CREATE and OPEN subprograms has been used by the implementation in order to express a set of exchange modes which characterize MARA files transfers.



The form string has the following syntax:

```
<form_parameter>      ::=      <form_modality>  {,<form_modality>}
<form_modality>        ::=      <positive_form_modality>
                           | <negative_form_modality>
```

```
<positive_form_modality> ::=
    LINE_FOLDED
    LINE_EDIT
    DOUBLE_BUFFER
    ECHO
    TIMED
    FULL_DUPLEX
    TERMINATOR
    SHARE_IN
    SHARE_OUT
    SHARE_INOUT
```

```
<negative_form_modality> ::= NO <positive_form_modality>
```

USE\_ERROR exception is raised if FORM string does not observe this syntax.

Modes expressed in small letters are considered as illegal.

The various operating modes which can be expressed through the form string, and the relative implementation behaviour are listed below.

#### 11.1.6.1 Line Folding

It allows to select the line terminator coding.

It is feasible with the FORM string [NO]LINE\_FOLDED option.

NOLINE\_FOLDED value codifies the line terminator with <ASCII.CR> control character.

LINE\_FOLDED value codifies the line terminator with <ASCII.CR,ASCII.LF> control characters sequence.

#### 11.1.6.2 Line Editing Mode

It allows the activation of the line editing as it is defined in [SDD] for input modes. In addition, it allows the activation of page terminator recognition codified by the implementation with the Form Feed control character which corresponds to <ASCII.FF> code. It is feasible with [NO]LINE\_EDIT option.

LINE\_EDIT value activates the SDD line edit, and deactivates the page terminator recognition. NOLINE\_EDIT value activates the page terminator recognition, and deactivates the SDD line editing.

In `LINE_EDIT` mode the input requests are terminated by line terminator acknowledge. The `TEXT_IO.SKIP_PAGE` send the `USE_ERROR` exception and "a page terminator never immediately follows a line terminator" is assumed by the `TEXT_IO.SKIP_LINE` subprogram.

#### **11.1.6.3 Character Echo**

It allows the programming of the SDD driver for echo mode as it is described in [SDD].

It is feasible through [NO]ECHO option.

#### **11.1.6.4 Line Terminator**

It allows the programming of the SDD driver for line terminator recognition.

It is feasible through [NO]TERMINATOR option.

#### **11.1.6.5 Character Full Duplex Exchange**

It allows the programming of the SDD driver to exchange a character in full duplex.

It is feasible through [NO]FULL\_DUPLEX option.

FULL\_DUPLEX option allows the output of one byte at most without interrupting any read operations in progress.

#### **11.1.6.6 Double Buffer Exchange Mode**

It allows to uncouple exchange cycles at the two sides of the path associated to the file by using two buffers which support the exchange.

It is feasible through [NO]DOUBLE\_BUFFER option.

These buffers size is a parameter of the static implementation configuration, and acts at node level.

#### **11.1.6.7 External Files Sharing**

It allows to specify what reading, writing, and updating modes can be shared with the mode expressed by `CREATE` and `OPEN` subprograms `MODE` parameter.

It is feasible through [NO]SHARE\_IN, [NO]SHARE\_OUT, and [NO]SHARE\_INOUT options.

It must be observed that this external file sharing control involves various `FILE_TYPE` objects use. These options have no effect on contemporary accesses performed by various tasks to the external file

through the exclusive use of a single file type object. As to this second problem, refer to Transportability notes section in this paper.

An external file sharing, if it is not appropriately controlled, can cause an unexpected sequencing of objects values stored in the file. This is not necessarily a problem when this sequencing is unessential (teletypes). On the other hand, it can be disastrous when a particular piece of information position in the file is the access key to the piece of information itself. As a consequence, a more careful use of the FORM string compared with sharing options use is recommended.

#### 11.1.6.8 Path Time Out

It allows to limit the waiting time related to OPEN and CREATE files requests to the time value specified in BASIC\_IO\_CONFIGURATION package, i.e., to system default.

It is feasible through [NO]TIMED option.

Files whose access has been NOTIMED required, will activate a null waiting.

- NOTE: Waiting time control on OPEN and CREATE requests exclusively operates on Attach and Create path requests addressed to path manager (see [KER]). Any waiting on the following path open request is not included in waiting time required.

Waiting time handling on the path open depends on the path server involved. Consequently, refer to the relative Mara documents.

The USE\_ERROR exception is raised at time-out expired when a non condivable file access is attempt. The non condivable condition is imposed by previous access.

The USE\_ERROR exception is raised also when a non multiuser path server managed device access is attempt.

The DEVICE\_ERROR exception is raised in all other cases.

#### 11.2 TEXT\_IO Standard Devices Control

Implementation associates TEXT\_IO.STANDARD\_OUTPUT and TEXT\_IO.STANDARD\_INPUT to MARA files defined respectively by :CO: and :CI: logic names. The relative physical name are required to the operating system by means of ProcessParms primitive invoking (see [KER] Process Management). This primitive returns to any parameters egment passed by the activator agent in which these names will be compiled in accordance with the modes described in [SPI].

The parameters buffer organization adopted by the implementation is the following:

```
type path_name is new plm_string;

type process_buffer is
record
reserved      :   path_manager_header;
prog_id       :   unsignedword;
spi_buf_id_1  :   byte;
spi_buf_id_2  :   byte;
ci_name       :   path_name;
lp_name       :   path_name;
co_name       :   path_name;
spi_buf_fac   :   byte;
console_info  :   plm_string;
end;
```

PATH\_NAME fields are organized in the memory as PLM-286 strings (see [PLM]). The implementation interpretes these fields in the succession previously mentioned. Thus, these fields position is essential in order to associate correctly the path names to the relative standard devices.

Spi\_buf\_id\_1 and spibufid2 fields identify the process buffer type. At present, only the buffer identified by the couple OFFH, 1 (or 9) - which has the structure above shown - is supported.

Spi\_buf\_fac field is not used, but it must be always present. It fills 8 successive memory bits (byte PLM-286).

CONSOLE\_INFO field is optional, and can be used to define again :CI: and :CO: values, and to define the values which must be associated to the two :FI: and :FO: strings that, in every respect, are the two logic names of :CI: and :CO: format strings.

This field is a PLM string whose first byte reports - as a consequence - the number of following characters included in the string. The following characters sequence must observe the syntax below:

```
<console_info> ::= [ <stuff> ]
                  - <console_token> { - <console_token> }
                  <terminator>

<stuff> ::= <sequence_of_char>

<console token> ::= :CI:= <sequence_of_char>
                  | :CO:= <sequence_of_char>
                  | :FI:= <sequence_of_char>
                  | :FO:= <sequence_of_char>

<sequence_of_char> ::= <ascii_character> (
<ascii_character> )

<terminator> ::= ascii.cr  ascii.lf
```

Mutual position of the various tokens is not essential. If a token is present more times, the last instance is the one considered. The first token is found immediately after the first - met. Whatever follows = until the next - , or at the end of the line, is considered part of a <sequence\_of\_char>.

It is the Ada program activator agent to compile such information in the parameters segment, and pass it then to application according to modes described in [KER] program loader.

In case Mini Session Monitor (see [MSM] ) is an Ada program activator agent, CONSOLE\_INFO field reports, according to described modes, command tail which accompanies program activation request. Such field can be used, therefore, to redefine path names associated to :CI: AND :CO: and relative form string.

CONSOLE\_INFO field absence in the parameters segment associates to :FI: and :FO: form string, system values defined through BASIC\_IO generic package.

The absence of parameters segment or its erroneous compilation cause NAME\_ERROR exception to be raised.

Such exception is handled during TEXT\_IO package processing allowing program processing continuation. In such circumstances the use of subprograms operating on default files cause STATUS\_ERROR exception to be raised. Consequently, the use of DEFAULT\_DEVICES generic package is recommended which allows to associate, to default devices, devices or files defined by the user.

Some examples of Ada programs invocation operated through Mini Session Monitor follow.

```
- MY.LTL          -:CI:=:FMS:MARALAB/COMMAND.CCC-:FI:=NOSHARE_OUT
- YOUR.LTL        -:FI:=SHARE_OUT-:FO:=NOTIMED
```

MY.LTL invocation addresses TEXT\_IO.STANDARD\_INPUT to :FMS:MARALAB/COMMAND.CCC. file Form string for such input prevents TEXT\_IO.OUT\_FILE mode application on such file. TEXT\_IO.STANDARD\_OUTPUT will be defined by Mini Session Monitor policies, while the relative form string is defined by current system default through generic BASIC\_IO.

YOUR.LTL invocation allows to share output on the path associated by MSM to :CI:, and not to require path timeout for access to :CI:. Such paths are defined by MSM policies, and they typically coincide with the terminal from which program activation is required.

### 11.3 I/O Package Specifications

The specifications of the standard I/O packages follow:

#### 11.3.1 IO\_EXCEPTIONS

```

-----
--
-- Date          20 April 1983
--
-- Programmer    Peter Haff
--
-- Project       Portable Ada Programming System
--
-- Module        IO_EXCPS.ADA
--
-- Description    Specification of package IO_EXCEPTIONS.
--               RM section 14.5.
--
-- Changes       Initial version 20 April 1983
--
-- DDC-I Ada (R) Compiler System (TM)
-- Copyright (C) 1984
-- DDC International A/S
-- All Rights Reserved
-- TO BE TREATED IN CONFIDENCE
--
-- (R) Ada is a registered trademark of the U.S. Government,
--     Ada Joint Program Office.
--
-- (TM) DDC-I Ada Compiler System is a trademark of DDC International
-- A/S.
-----

```

```
pragma page;
```

```
package IO_EXCEPTIONS is
```

```
-- The order of the following declarations must NOT be changed:
```

```

STATUS_ERROR : exception;
MODE_ERROR   : exception;
NAME_ERROR   : exception;
USE_ERROR    : exception;
DEVICE_ERROR : exception;
END_ERROR    : exception;
DATA_ERROR   : exception;
LAYOUT_ERROR : exception;

```

```
end IO_EXCEPTIONS;
```

### 11.3.2 BASIC\_IO\_TYPES

```

-- Date          19 March 1986
--
-- Programmer     Knud Joergen Kirkegaard
--
-- Project        DDC Ada Compiler System
--                VAX-11 hosted
--
-- Module         BASIC_IO_TYPES (spec)
--
-- Description     Ada Input-Output System for VAX/VMS
--
--                Associated Documents:
--                DDC 5118/RPT/20, issue 2
--                DDC 118/RPT/13, issue 4
--
--                For Ada releases >= *.*
--
-- Changes        Initial version 19 March 1986
--
-- Copyright 1986 by Dansk Datamatik Center (DDC).
-- This program as well as any listing thereof may not
-- be reproduced in any form without prior permission
-- in writing from DDC.
--

```

```

-----
pragma page;
with system;

```

```

package basic_io_types is

```

```

    subtype io_kind is integer;

```

```

    -- io_kind is used in the create and open procedures and indicates:
    --          0 : sequential file
    --          1 : direct file
    --          2 : text file
    --          3 : variable file (this is not used by the standard
packages)

```

```

    type file_mode is range 0 .. 2;

```

```

    -- file_mode indicates the mode of the file:
    --          for sequential and text files:
    --              0 : in_file
    --              1 : out_file
    --
    --          for direct and variable files:
    --              0 : in_file
    --              1 : inout_file
    --              2 : out_file

```

```

    subtype key is long_integer;

```

```

-- key is used by variable i/o to indicate the position in a file
-- this may be implementation dependent

type file_type is access integer;

type count is range 0 .. long_integer'last;

subtype positive_count is count range 1 .. count'last;

subtype pointer is system.address;

-- pointer is used in the read and write procedures to indicate a
data area

end basic_io_types;

```

### 11.3.3 TEXT\_IO

```

-----
-----
--
-- Date          31 October 1983
--
-- Programmer    Soeren Prehn (, Knud Joergen Kirkegaard)
--
-- Project       Portable Ada Programming System
--               Input-Output
--
-- Module        TEXTIOS.ADA
--
-- Description    Specification of Package TEXT_IO,
--               Ada LRM (jan 83) 14.3.10
--
-- Changes       Initial version 31 October 1983
--               Adapted to KAPSE interface specification 17 April
1986
--
-- DDC-I Ada Compiler System (TM)
-- DACS for VAX/VMS
-- DDC-I PROPRIETARY INFORMATION:
-- Copyright (C) 1984 DDC International A/S.
-- All rights reserved. This material contains unpublished
-- trade secret information from DDC International A/S.
-- TO BE TREATED IN CONFIDENCE
--
-- (TM) DDC-I Ada Compiler System is a trademark of DDC International
A/S.
--
-----
-----

pragma page;
with BASIC_IO_TYPES;

```



```
with IO_EXCEPTIONS;
package TEXT_IO is
```

```
    type FILE_TYPE is limited private;
```

```
    type FILE_MODE is (IN_FILE, OUT_FILE);
```

```
    type COUNT is range 0 .. LONG_INTEGER'LAST;
```

```
    subtype POSITIVE_COUNT is COUNT range 1 .. COUNT'LAST;
```

```
    UNBOUNDED: constant COUNT:= 0; -- line and page length
```

```
    subtype FIELD          is INTEGER range 0 .. 67; -- max. size of
an integer output field
```

```
    subtype NUMBER_BASE    is INTEGER range 2 .. 16; -- 2#....#
```

```
    type TYPE_SET is (LOWER_CASE, UPPER_CASE);
```

```
pragma PAGE;
```

```
-- File Management
```

```
    procedure CREATE (FILE : in out FILE_TYPE;
                      MODE : in FILE_MODE := OUT_FILE;
                      NAME : in STRING := "";
                      FORM : in STRING := ""
                      );
```

```
    procedure OPEN (FILE : in out FILE_TYPE;
                   MODE : in FILE_MODE;
                   NAME : in STRING;
                   FORM : in STRING := ""
                   );
```

```
    procedure CLOSE (FILE : in out FILE_TYPE);
```

```
    procedure DELETE (FILE : in out FILE_TYPE);
```

```
    procedure RESET (FILE : in out FILE_TYPE; MODE : in FILE_MODE);
```

```
    procedure RESET (FILE : in out FILE_TYPE);
```

```
    function MODE (FILE : in FILE_TYPE) return FILE_MODE;
```

```
    function NAME (FILE : in FILE_TYPE) return STRING;
```

```
    function FORM (FILE : in FILE_TYPE) return STRING;
```

```
    function IS_OPEN(FILE : in FILE_TYPE) return BOOLEAN;
```

```
pragma PAGE;
```

```
-- Control of default input and output files
```

```
    procedure SET_INPUT (FILE : in FILE_TYPE);
```

```
    procedure SET_OUTPUT (FILE : in FILE_TYPE);
```

```
    function STANDARD_INPUT return FILE_TYPE;
```

```
    function STANDARD_OUTPUT return FILE_TYPE;
```

```
    function CURRENT_INPUT return FILE_TYPE;
```

```
    function CURRENT_OUTPUT return FILE_TYPE;
```

```

pragma PAGE;
-- specification of line and page lengths

procedure SET_LINE_LENGTH (FILE : in FILE_TYPE; TO : in COUNT);
procedure SET_LINE_LENGTH (TO : in COUNT);

procedure SET_PAGE_LENGTH (FILE : in FILE_TYPE; TO : in COUNT);
procedure SET_PAGE_LENGTH (TO : in COUNT);

function LINE_LENGTH (FILE : in FILE_TYPE) return COUNT;
function LINE_LENGTH return COUNT;

function PAGE_LENGTH (FILE : in FILE_TYPE) return COUNT;
function PAGE_LENGTH return COUNT;

pragma PAGE;
-- Column, Line, and Page Control

procedure NEW_LINE (FILE : in FILE_TYPE; SPACING : in
POSITIVE_COUNT := 1);
procedure NEW_LINE (SPACING : in
POSITIVE_COUNT := 1);

procedure SKIP_LINE (FILE : in FILE_TYPE; SPACING : in
POSITIVE_COUNT := 1);
procedure SKIP_LINE (SPACING : in
POSITIVE_COUNT := 1);

function END_OF_LINE (FILE : in FILE_TYPE) return BOOLEAN;
function END_OF_LINE return BOOLEAN;

procedure NEW_PAGE (FILE : in FILE_TYPE);
procedure NEW_PAGE ;

procedure SKIP_PAGE (FILE : in FILE_TYPE);
procedure SKIP_PAGE ;

function END_OF_PAGE (FILE : in FILE_TYPE) return BOOLEAN;
function END_OF_PAGE return BOOLEAN;

function END_OF_FILE (FILE : in FILE_TYPE) return BOOLEAN;
function END_OF_FILE return BOOLEAN;

procedure SET_COL (FILE : in FILE_TYPE; TO : in
POSITIVE_COUNT);
procedure SET_COL (TO : in
POSITIVE_COUNT);

procedure SET_LINE (FILE : in FILE_TYPE; TO : in
POSITIVE_COUNT);
procedure SET_LINE (TO : in
POSITIVE_COUNT);

function COL (FILE : in FILE_TYPE) return
POSITIVE_COUNT;
function COL return

```

```

POSITIVE_COUNT;

    function      LINE                      (FILE : in FILE_TYPE) return
POSITIVE_COUNT;
    function      LINE                      return
POSITIVE_COUNT;

    function      PAGE                      (FILE : in FILE_TYPE) return
POSITIVE_COUNT;
    function      PAGE                      return
POSITIVE_COUNT;

pragma PAGE;
    -- Character Input-Output

    procedure GET  (FILE : in FILE_TYPE; ITEM : out CHARACTER);
    procedure GET  (                      ITEM : out CHARACTER);
    procedure PUT  (FILE : in FILE_TYPE; ITEM : in CHARACTER);
    procedure PUT  (                      ITEM : in CHARACTER);

    -- String Input-Output

    procedure GET  (FILE : in FILE_TYPE; ITEM : out STRING);
    procedure GET  (                      ITEM : out STRING);
    procedure PUT  (FILE : in FILE_TYPE; ITEM : in STRING);
    procedure PUT  (                      ITEM : in STRING);

    procedure GET_LINE (FILE : in FILE_TYPE; ITEM : out STRING; LAST
: out NATURAL);
    procedure GET_LINE (                      ITEM : out STRING; LAST
: out NATURAL);
    procedure PUT_LINE (FILE : in FILE_TYPE; ITEM : in STRING);
    procedure PUT_LINE (                      ITEM : in STRING);

pragma PAGE;
    -- Generic Package for Input-Output of Integer Types

generic
    type NUM is range <>;
package INTEGER_IO is

    DEFAULT_WIDTH : FIELD      := NUM'WIDTH;
    DEFAULT_BASE  : NUMBER_BASE :=      10;

    procedure GET  (FILE : in FILE_TYPE; ITEM : out NUM; WIDTH :
in FIELD := 0);
    procedure GET  (                      ITEM : out NUM; WIDTH :
in FIELD := 0);

    procedure PUT  (FILE : in FILE_TYPE;
                    ITEM : in NUM;
                    WIDTH : in FIELD := DEFAULT_WIDTH;
                    BASE : in NUMBER_BASE := DEFAULT_BASE);
    procedure PUT  (ITEM : in NUM;
                    WIDTH : in FIELD := DEFAULT_WIDTH;

```

```

        BASE : in NUMBER_BASE := DEFAULT_BASE);

    procedure GET (FROM : in STRING; ITEM : out NUM; LAST : out
POSITIVE);
    procedure PUT (TO : out STRING;
        ITEM : in NUM;
        BASE : in NUMBER_BASE := DEFAULT_BASE);

end INTEGER_IO;

pragma PAGE;
-- Generic Packages for Input-Output of Real Types

generic
    type NUM is digits <>;
package FLOAT_IO is

    DEFAULT_FORE : FIELD := 2;
    DEFAULT_AFT : FIELD := NUM'DIGITS - 1;
    DEFAULT_EXP : FIELD := 3;

    procedure GET (FILE : in FILE_TYPE; ITEM : out NUM; WIDTH : in
FIELD := 0);
    procedure GET (ITEM : out NUM; WIDTH : in
FIELD := 0);

    procedure PUT (FILE : in FILE_TYPE;
        ITEM : in NUM;
        FORE : in FIELD := DEFAULT_FORE;
        AFT : in FIELD := DEFAULT_AFT;
        EXP : in FIELD := DEFAULT_EXP);

    procedure PUT (ITEM : in NUM;
        FORE : in FIELD := DEFAULT_FORE;
        AFT : in FIELD := DEFAULT_AFT;
        EXP : in FIELD := DEFAULT_EXP);

    procedure GET (FROM : in STRING; ITEM : out NUM; LAST : out
POSITIVE);
    procedure PUT (TO : out STRING;
        ITEM : in NUM;
        AFT : in FIELD := DEFAULT_AFT;
        EXP : in FIELD := DEFAULT_EXP);

end FLOAT_IO;

pragma PAGE;
generic
    type NUM is delta <>;
package FIXED_IO is

    DEFAULT_FORE : FIELD := NUM'FORE;
    DEFAULT_AFT : FIELD := NUM'AFT;
    DEFAULT_EXP : FIELD := 0;

    procedure GET (FILE : in FILE_TYPE; ITEM : out NUM; WIDTH : in
FIELD := 0);

```

```

    procedure GET (
        FIELD := 0);
        ITEM : out NUM; WIDTH : in

    procedure PUT (FILE : in FILE_TYPE;
        ITEM : in NUM;
        FORE : in FIELD := DEFAULT_FORE;
        AFT : in FIELD := DEFAULT_AFT;
        EXP : in FIELD := DEFAULT_EXP);

    procedure PUT (ITEM : in NUM;
        FORE : in FIELD := DEFAULT_FORE;
        AFT : in FIELD := DEFAULT_AFT;
        EXP : in FIELD := DEFAULT_EXP);

    procedure GET (FROM : in STRING; ITEM : out NUM; LAST : out
    POSITIVE);
    procedure PUT (TO : out STRING;
        ITEM : in NUM;
        AFT : in FIELD := DEFAULT_AFT;
        EXP : in FIELD := DEFAULT_EXP);

end FIXED_IO;

pragma PAGE;
-- Generic Package for Input-Output of Enumeration Types

generic
    type ENUM is (<>);
package ENUMERATION_IO is

    DEFAULT_WIDTH : FIELD := 0;
    DEFAULT_SETTING : TYPE_SET := UPPER_CASE;

    procedure GET (FILE : in FILE_TYPE; ITEM : out ENUM);
    procedure GET (
        ITEM : out ENUM);

    procedure PUT (FILE : in FILE_TYPE;
        ITEM : in ENUM;
        WIDTH : in FIELD := DEFAULT_WIDTH;
        SET : in TYPE_SET := DEFAULT_SETTING);

    procedure PUT (ITEM : in ENUM;
        WIDTH : in FIELD := DEFAULT_WIDTH;
        SET : in TYPE_SET := DEFAULT_SETTING);

    procedure GET (FROM : in STRING; ITEM : out ENUM; LAST : out
    POSITIVE);
    procedure PUT (TO : out STRING;
        ITEM : in ENUM;
        SET : in TYPE_SET := DEFAULT_SETTING);

end ENUMERATION_IO;

pragma PAGE;
-- Exceptions

```

```

STATUS_ERROR : exception renames IO_EXCEPTIONS.STATUS_ERROR;
MODE_ERROR   : exception renames IO_EXCEPTIONS.MODE_ERROR;
NAME_ERROR   : exception renames IO_EXCEPTIONS.NAME_ERROR;
USE_ERROR    : exception renames IO_EXCEPTIONS.USE_ERROR;
DEVICE_ERROR : exception renames IO_EXCEPTIONS.DEVICE_ERROR;
END_ERROR    : exception renames IO_EXCEPTIONS.END_ERROR;
DATA_ERROR   : exception renames IO_EXCEPTIONS.DATA_ERROR;
LAYOUT_ERROR : exception renames IO_EXCEPTIONS.LAYOUT_ERROR;

```

```

pragma page;
private

```

```

type FILE_BLOCK_TYPE is new BASIC_IO_TYPES.FILE_TYPE;

```

```

type FILE_OBJECT_TYPE is
  record
    IS_OPEN      : BOOLEAN           := FALSE;
    FILE_BLOCK   : FILE_BLOCK_TYPE;
  end record;

```

```

type FILE_TYPE is access FILE_OBJECT_TYPE;

```

```

end TEXT_IO;

```

#### 11.3.4 SEQUENTIAL\_IO

```

-----
--
-- Date          20 April 1983
--
-- Programmer    Peter Haff (, Soeren Prehn, Knud Joergen Kirkegaard)
--
-- Project       Portable Ada Programming System
--
-- Module        SEQIOS.ADA
--
-- Description    Specification of package SEQUENTIAL_IO.
--               LRM (Jan 83) section 14.2.3
--
-- Changes       Initial version 20 April 1983
--
-- DDC-I Ada (R) Compiler System (TM)
-- Copyright (C) 1984
-- DDC International A/S
-- All Rights Reserved
-- TO BE TREATED IN CONFIDENCE
--
-- (R) Ada is a registered trademark of the U.S. Government,
--   Ada Joint Program Office.
--
-- (TM) DDC-I Ada Compiler System is a trademark of DDC International
-- A/S.
--

```

-----  
-----  
pragma PAGE;

with IO\_EXCEPTIONS;  
with BASIC\_IO\_TYPES;

generic

type ELEMENT\_TYPE is private;

package SEQUENTIAL\_IO is

type FILE\_TYPE is limited private;

type FILE\_MODE is (IN\_FILE, OUT\_FILE);

pragma PAGE;

-- File management

procedure CREATE (FILE : in out FILE\_TYPE;  
                  MODE : in      FILE\_MODE := OUT\_FILE;  
                  NAME : in      STRING   := "";  
                  FORM : in      STRING   := "");

procedure OPEN  (FILE : in out FILE\_TYPE;  
                  MODE : in      FILE\_MODE;  
                  NAME : in      STRING;  
                  FORM : in      STRING := "");

procedure CLOSE (FILE : in out FILE\_TYPE);

procedure DELETE (FILE : in out FILE\_TYPE);

procedure RESET (FILE : in out FILE\_TYPE;  
                  MODE : in      FILE\_MODE);

procedure RESET (FILE : in out FILE\_TYPE);

function MODE  (FILE : in FILE\_TYPE) return FILE\_MODE;

function NAME  (FILE : in FILE\_TYPE) return STRING;

function FORM  (FILE : in FILE\_TYPE) return STRING;

function IS\_OPEN (FILE : in FILE\_TYPE) return BOOLEAN;

pragma PAGE;

-- input and output operations

procedure READ  (FILE : in      FILE\_TYPE;  
                  ITEM :      out ELEMENT\_TYPE);

```

procedure WRITE (FILE : in FILE_TYPE;
                 ITEM : in ELEMENT_TYPE);

function END_OF_FILE(FILE : in FILE_TYPE) return BOOLEAN;

pragma PAGE;
-- exceptions

STATUS_ERROR : exception renames IO_EXCEPTIONS.STATUS_ERROR;
MODE_ERROR   : exception renames IO_EXCEPTIONS.MODE_ERROR;
NAME_ERROR   : exception renames IO_EXCEPTIONS.NAME_ERROR;
USE_ERROR    : exception renames IO_EXCEPTIONS.USE_ERROR;
DEVICE_ERROR : exception renames IO_EXCEPTIONS.DEVICE_ERROR;
END_ERROR    : exception renames IO_EXCEPTIONS.END_ERROR;
DATA_ERROR   : exception renames IO_EXCEPTIONS.DATA_ERROR;

pragma PAGE;
private

type FILE_TYPE is new BASIC_IO_TYPES.FILE_TYPE;

end SEQUENTIAL_IO;

```

### 11.3.5 DIRECT\_IO

```

-----
-----
--
-- Date          20 April 1983
--
-- Programmer     Peter Haff (, Soeren Prehn, Knud Joergen Kirkegaard)
--
-- Project        Portable Ada Programming System
--
-- Module         DIR_IO.ADA
--
-- Description     Specification of package DIRECT_IO.
--                LRM (jan 83) section 14.2.5.
--
-- Changes        Initial version 20 April 1983
--                31 OCT 1983: FILE_TYPE made private. /SP
--                21 March 1986: Adapted to KAPSE interface
specification.
--
-- DDC-I Ada (R) Compiler System (TM)
-- Copyright (C) 1984
-- DDC International A/S
-- All Rights Reserved
-- TO BE TREATED IN CONFIDENCE
--
-- (R) Ada is a registered trademark of the U.S. Government,
-- Ada Joint Program Office.

```



```
--
-- (TM) DDC-I Ada Compiler System is a trademark of DDC International
A/S.
--
```

```
-----
pragma PAGE;
with IO_EXCEPTIONS;
with BASIC_IO_TYPES;
```

```
generic
```

```
    type ELEMENT_TYPE is private;
```

```
package DIRECT_IO is
```

```
    type FILE_TYPE is limited private;
```

```
    type FILE_MODE is (IN_FILE, INOUT_FILE, OUT_FILE);
```

```
    type COUNT is range 0..LONG_INTEGER'LAST;
```

```
    subtype POSITIVE_COUNT is COUNT range 1..COUNT'LAST;
```

```
pragma PAGE;
```

```
-- File management
```

```
    procedure CREATE (FILE : in out FILE_TYPE;
                      MODE : in FILE_MODE := INOUT_FILE;
                      NAME : in STRING := "";
                      FORM : in STRING := "");
```

```
    procedure OPEN (FILE : in out FILE_TYPE;
                   MODE : in FILE_MODE;
                   NAME : in STRING;
                   FORM : in STRING := "");
```

```
    procedure CLOSE (FILE : in out FILE_TYPE);
```

```
    procedure DELETE (FILE : in out FILE_TYPE);
```

```
    procedure RESET (FILE : in out FILE_TYPE;
                    MODE : in FILE_MODE);
```

```
    procedure RESET (FILE : in out FILE_TYPE);
```

```
    function MODE (FILE : in FILE_TYPE) return FILE_MODE;
```

```
    function NAME (FILE : in FILE_TYPE) return STRING;
```

```
    function FORM (FILE : in FILE_TYPE) return STRING;
```

```
    function IS_OPEN (FILE : in FILE_TYPE) return BOOLEAN;
```

```
pragma PAGE;
-- input and output operations
```

```

procedure READ (FILE : in FILE_TYPE;
                ITEM : out ELEMENT_TYPE;
                FROM : in POSITIVE_COUNT);
procedure READ (FILE : in FILE_TYPE;
                ITEM : out ELEMENT_TYPE);

procedure WRITE (FILE : in FILE_TYPE;
                 ITEM : in ELEMENT_TYPE;
                 TO : in POSITIVE_COUNT);
procedure WRITE (FILE : in FILE_TYPE;
                 ITEM : in ELEMENT_TYPE);

procedure SET_INDEX (FILE : in FILE_TYPE;
                    TO : in POSITIVE_COUNT);

function INDEX (FILE : in FILE_TYPE) return POSITIVE_COUNT;

function SIZE (FILE : in FILE_TYPE) return COUNT;

function END_OF_FILE (FILE : in FILE_TYPE) return BOOLEAN;
```

```
pragma PAGE;
-- exceptions
```

```

STATUS_ERROR : exception renames IO_EXCEPTIONS.STATUS_ERROR;
MODE_ERROR   : exception renames IO_EXCEPTIONS.MODE_ERROR;
NAME_ERROR   : exception renames IO_EXCEPTIONS.NAME_ERROR;
USE_ERROR    : exception renames IO_EXCEPTIONS.USE_ERROR;
DEVICE_ERROR : exception renames IO_EXCEPTIONS.DEVICE_ERROR;
END_ERROR    : exception renames IO_EXCEPTIONS.END_ERROR;
DATA_ERROR   : exception renames IO_EXCEPTIONS.DATA_ERROR;
```

```
pragma PAGE;
private
```

```

    type FILE_TYPE is new BASIC_IO_TYPES.FILE_TYPE;

end DIRECT_IO;
```

## 12 Basic I/O configurability

The present section lists configurability elements of Ada basic I/O support defined by BASIC\_LEXICAL\_IO, BASIC\_TEXT\_IO and BASIC\_COMMON\_IO packages (see [UserGuide]).

As basic I/O directly supports Ada standard I/O, configurability features described here can be transitively transported to standard I/O context.

Support configurability concerns control of system resources committed for a Mara file access, as well as control of access modes to a file expressed by FORM string (see [BIO]).

Both control types are feasible both at single program level and at system level. The latter acts as default for program level.

Program configurability is feasible through generic packages listed below. Such library units specification is reported in Appendix A.

System configurability is expressed both through SCL language, in ADABIO factory product description (see [ADABIO]), and through the use of BASIC\_IO generic package.

## 12.1 Program Configurability

Ada basic I/O support presents some configurability characteristics that application program can control using BASIC\_IO\_CONFIGURATION, DEFAULT\_FORM and DEFAULT\_DEVICES generic packages.

### 12.1.1 DEFAULT\_DEVICES Generic Package

[...] details choices operated by the implementation in associating appropriate external files to TEXT\_IO.STANDARD\_INPUT and TEXT\_IO.STANDARD\_OUTPUT standard devices and therefore to default ones (see [LRM] ch 14.3). In case system choices are not applicable, the user can provide new values to default devices through DEFAULT\_DEVICES generic package.

Such values act at application program level. Other node programs are not influenced by the new choice.

```
with default_devices;
package my_devices is new default_devices (
  output_name      => :D:0007 ,
  input_name       => :D:0007 ,
  input_form       => SHARE_OUT ,
  output_form      => NOTIMED);
with my_devices;
procedure my_program is ... end;
```

The example shows a possible use of such generic, operated at program library level.

After instantiation of such package has been processed, :D:0007 path name refers to device to be used as current default input and current default output. As regards input form string, it is allowed to share output mode. As regards output form string, a null wait is required on relative attach path request.

If such package is instantiated more times within a program, effective choice will depend on processing order fixed by compiler and linker of the various instantiations (see [LRM] ch. 10), according to

programmed  
dependence between the various program units.

NOTE: The use of such generic is useful only when TEXT\_IO.STANDARD\_INPUT and TEXT\_IO.STANDARD\_OUTPUT correspond to no external file. This occurs in case of absence or erroneous compilation of parameters segment passed to an Ada application main program by environment agent that this activates.

DEFAULT\_DEVICES instantiation in presence of parameters segment, corresponds to a no operation to the full.

In other terms, DEFAULT\_DEVICES:

- prevents TEXT\_IO.STANDARD\_INPUT and TEXT\_IO.STANDARD\_OUTPUT modification. These files are always associated to :CI: and :CO:, whatever their value is;

- prevents TEXT\_IO.CURRENT\_INPUT and TEXT\_IO.CURRENT\_OUTPUT redirection on new default values whenever current defaults are associated to some external file.

If TEXT\_IO.CURRENT\_INPUT and TEXT\_IO.CURRENT\_OUTPUT redirection is needed, you are invited to use TEXT\_IO relative subprograms explicitly.

#### 12.1.2 DEFAULT\_FORM Generic Package

An application user can arrange default values limited by program context through DEFAULT\_FORM generic package use.

```
with default_form;
package my_form is new default_form (      DOUBLE_BUFFER &
,LINE_FOLDED &
,NOLINE_EDIT &
,SHARE_IN &
,NOSHARE_OUT &
,NOSHARE_INOUT &
,ECHO &
,TERMINATOR, &
,TIMED );
```

```
with my_form:
procedure my_program is ... end;
```

The example shows a possible use of such generic, operated at program library level.

After such package instantiation has been processed, the indicated string value replaces previous values defined by previous instantiations, that is system ones.

If such package is instantiated more times within a program, effective

choice will depend on various instantiations processing order established by compiler and linker (see [LRM] ch. 10), according to programmed dependences between the various program units.

### 12.1.3 BASIC\_IO\_CONFIGURATION Generic Package

BASIC\_IO\_CONFIGURATION generic package allows to fix the following quantities in the program context:

- increase with respect to the current value of connections number to be operated towards external files: NUMBER\_OF\_FILES;
- time, expressed with CALENDAR.DAY\_DURATION typology, you are ready to wait for on OPEN or CREATE requests to complete them: TIME\_OUT.
- value of exchange buffers width on accessed files: BUFFER\_SIZE.

```
with basic_io_configuration;  
package my_configuration is basic_io_configuration (  
  buffer_size      =>      3000;  
  time_out         =>      120.0;  
  number_of_files; =>      15)
```

The example indicates to process with maximum 15 files which are opened contemporarily. As support memory to the fifteen connections is accorded by Kernel and paged up with 128 storage units pages, and as connection unitary support is not commensurable with 128, it may occur that a higher number of files be actually opened.

To allocate necessary support memory, implementation adopts the minimum number of segments. Such segments are required together with those OPEN and CREATE requests which exhaust current segment.

New segments request is disabled beyond the limit declared by BASIC\_IO\_CONFIGURATION package instantiation. In this circumstance, STORAGE\_ERROR exception is raised on request of extra-connections with respect to declared ones.

Segment unavailability causes STORAGE\_ERROR exception to be raised even below the declared limit.

A careful use of such package allows to dimension memory segments width at best on grounds of requisites declared by an application program, though it does not ensure memory availability which is however necessary to support the number of required connections. So you are encouraged to use it !

The other parameters require a waiting time which does not exceed two minutes for each access request on a MARA file as well as a 3000 storage units wide exchange with buffer.

Processing order will decide which values will be active between one instantiation and the following, as for the other generic supports in case of multiple instantiations of such package.

Generic package specification does not provide default values. Therefore, it is necessary to define a complete triple for the three parameters on each instantiation, otherwise it is impossible to compile the instantiation.

## 12.2 Static Configurability

### 12.2.1 DATA clause

Though DATA clause of SCL description of support to Ada basic i/o (see [ADABIO]), it is possible to dimension default values for parameters defined by BASIC\_IO\_CONFIGURATION package.

```
DATA RTS.OBJ
    MAX_NUMBER_OF_FILES      = WORD P_NUMBER$OF$FILES;
    MIN_TIME_OUT_LOW         = WORD P_TIME$OUT$LOW;
    MIN_TIME_OUT_HIGH        = WORD P_TIME$OUT$HIGH;
    SIZE_OF_BUFFERS          = WORD P_BUFFER$SIZE;
END DATA;
```

. . . .

```
INVOKE ADABIO (13,0000,7H,2500) REPEATED;
```

The example shows a static configuration which will afford access to no more than 13 contemporary connections to MARA files for each program executed in the node. Each connection will be temporized for no more than  $7 * 2 \text{ exp } (16)$  microseconds. The buffers exchanged on each connection will be 2500 storageunits (word) wide.

The use of BASIC\_IO\_CONFIGURATION package allows to replace program defaults with system values expressed by DATA clause.

The following unitary commitment of system resources for dimensioning of P\_NUMBER\$OF\$FILES parameter must be considered:

- 1 path;
- 2 path mailboxes;
- 1 + x path buffers with x in 1..2 according to single or double buffer operation;
- 1 about 70 local or nodal bytes + 22 nodal bytes;

Nodal bytes will be really such for nodes having nodal memory. On the contrary, they will be summed to those required by local memory.

### 12.2.2 BASIC\_IO Generic Package

BASIC\_IO generic package allows to configure ADABIO product with respect to form tail value, which must act as system default, and with respect to format tail value which must act for files referred by :CI: and :CO: logic names, to which implementation associates TEXT\_IO.STANDARD\_INPUT and TEXT\_IO.STANDARD\_OUTPUT.

```
with basic_io;
package fi_configuration is new basic_io;
```

```
with basic_io;
package user_configuration is new basic_io
  (form => NODOUBLE_BUFFER &
   ,LINE_FOLDED &
   ,NOLINE_EDIT &
   ,NOSHARE_IN &
   ,NOSHARE_OUT &
   ,NOSHARE_INOUT &
   ,ECHO &
   ,TERMINATOR &
   ,TIMED &
   ci-form => NOSHARE_OUT ,
   co_form => TIMED );
```

In the first example, reconfiguration package configures basic\_io support accepting generic default values (see app. A).

In the second case user\_configuration package configures support, assigning, as system default, the following operating modalities:

- double buffer modality
- files will be line-editable
- their line terminator will be codified with <cr,lf> ,
- no sharing form will be possible,
- echo on input operations will be operated
- line and page terminator will be recognized
- access requests will be subdued to time out.

Once chosen configuration has been compiled, it can be installed in the node generating an Ada program on it, whose SCL specification describes an invoked type program (see [GENRP]).

An example of the system generation related to BASIC\_IO configuration is described below.

Ada text, contained in NODAL\_ELABORATION.ADA

```
with basic_io;
package fi_configuration is new basic_io;
```

```
with fi_configuration;
procedure nodal_elaboration is
begin
```

```
    null;
end;
```

SCL text, contained in MYSYS.SCL

. . . .

program template Nodal\_Elaboration large

```
code PRIVATE
data PRIVATE NODAL;
```

module	:FMS:ADARTSA1/RTSDATA.OBJ	relocatable;
module	:FMS:ADABIOA1/BIODATA.OBJ	relocatable;
module	:Nodalelaboration	relocatable;
module	:FMS:DACS86A0/ROOT.LIB	relocatable;
module	:FMS:DACS86A0/RTHELP286.LIB	relocatable;
module	:FMS:KER286A0/ADAUS.LIB	relocatable;
relocatable;		
module	:FMS:ADABIOA1/BINDER286.LIB	relocatable;
module	:FMS:ADARTSA1/BINDER286.LIB	relocatable;
module	GATELBA	relocatable;

```
initial procedure Nodal_Elaboration
stacksegment nodal size = 2000H;
```

```
end program;
```

. . . .

```
function ada_container
privilege                2
function_segment         100
maxpriority              7
initial program nodal_elaboration;
```

```
invoke nodal_elaboration;
invoke adabio repeated;
```

```
end function;
```

. . . .



DCL text

```
. . .  
$      SCL 286 MYSYS.SCL  
  
. . .  
$      ADA286/LIBRARY   = MY_ALB.ALB  ADABIOA0:NODAL_ELABORATION.ADA  
$      ASSIGN/NOLOG MYSYS.GRA  ADA286_GRAPH  
$      ASSIGN/NOLOG  NODAL_ELABORATION.LNK  NODAL_ELABORATION  
$      ADA286/LINK/LIBR   =MY_ALB.ALB/OFD=[  ]    /    TEMPLATE  
NODAL_ELABORATION  
. . .  
$      GENIP MYSYS.GRA ASSIGN NODAL_ELABORATION=NODAL_ELABORATION.LTL  
$      ABS286 MYSYS.IPL  
. . .
```

Such command sequence will deposit NODAL\_ELABORATION.LTL file in current directory, which together with the other programs described, will represent the system initial program.

During initialization phase, kernel initialization process will activate nodal\_elaboration program, which has been declared function initial program in the present example.

At the end of this elaboration the function will be declared ready through a non visible call of SET\_FUNCTION\_READY kernel procedure. The continuation of system initialization is than possible.

### 12.3 Basic I/O Intrinsic Limits

Lexical elements reading requests through TEXT\_IO generic subprograms cannot exceed 128 characters.

## 13 [BIO] INPUT OUTPUT HANDLING

### 13.1 Devices handled by SDD AND SDS

SDD and SDS (see [SDD],[SDS]) classify devices into three conventional classes:

- teletype;
- printer;

- video;

Teletype typology is used for host computers and terminals input and output connections.

Printer typology provides for host computers and printers only output connections.

Video typology provides for video terminals handling able to organize information according to predetermined masks.

Of the three devices classes, video class is the only one which cannot suitably correspond to any of the three standard models provided by the language.

Such limitation is essentially due to positional information organization on video device and to the typological variety of objects that can be exchanged with it. Textio and directio models mix seems to be therefore the best model to control conventional videos. To define such mix lies outside the scope of this document.

Both Ada text and Ada sequential files are useful to model the exchange with teletype and printer type interactive devices and host computers.

Independent of the file model chosen, various read options control towards conventional teletypes defined in [SDD] is feasible exclusively by using the following form string modes:

- [NO]TERMINATOR;
- [NO]LINE\_EDIT;
- [NO]LINE\_FOLDED;
- [NO]ECHO;
- [NO]DOUBLE\_BUFFER;

In the following two sections some guide notes and examples are reported which allow to use form string for an interactive devices and host computers efficient control through Ada file standard models.

### 13.1.1 Interactive Devices

Ada text files are the most suitable and flexible model to control teletype and printer type devices connected respectively to terminals and physical printers.

Suitability depends on the nature of such devices exchanging an information which is structured in itself on character basic typology in ASCII coding.

Flexibility depends on the presence of INTEGER\_IO, EUMERATION\_IO, FLOAT\_IO and FIXED\_IO packages which allow on the one hand to output

integer, enumerative and real type values in the ASCII format, that is directly interpreted by the device; on the other hand to input directly from the device integer, enumerative and real values, subduing them to language lexical control.

Such flexibility is paid with less efficient exchanges with respect to exchanges based on more elementary character or string types. Anyway, exchanges of this kind compel to use less structured types and therefore less powerful operators, which finally lead to a less compact and therefore less efficient code.

A conventional teletype associated to an interactive device supports characters line concept, allowing to complete input operations to identify a terminating sequence defined during activation of associated peripheral device. Such performance is directly feasible through TERMINATOR modes in form string.

According to layout handling needs of a text file generated by an interactive device, implementation allows, through LINE\_EDIT option (see [FORM]), to read towards SDD and SDS or in the so called line edit mode, or in transparent reading mode with terminator (see [SDD]).

In line edit mode, conventional teletypes are not able to generate text file layout, essentially for two reasons.

First because any control character generated by the device and different from those used to carry out line edit functionalities, is removed from SDD driver. Ascii.ff character will never be brought back from SDD to an application software.

Secondly because of an automatic line scrolling operated on the last video device useful line with no signal interpretable as page closing corresponding to it in the device itself.

This implies that SKIPLINE service implementation, in such devices operated according to line edit modes, can omit actions relative to identification of a possible page terminator immediately following line terminator. This ensures a device wait limited to identification of line terminator alone.

If it is necessary to use an interactive device to generate a page text file, NOLINE\_EDIT mode can be requested. In such mode, implementation reads towards device in transparent with terminator mode. In this situation, SKIP\_LINE and SKIP\_PAGE reactivate ascii.ff character identification, questioning the device for a further character after a line terminator in the case of SKIP\_LINE, or of ascii.ff character in the case of SKIP\_PAGE.NOTE.

Because of further activated waits, one can think that implementation behaviour is wrong. This is not true. Such behaviour is expected in this kind of input towards conventional teletypes which are interactive devices.

### 13.1.2 Connections Towards Host Computer

Both text files and sequential files can be used to control the input of a conventional teletype connected to a host computer.

Ada sequential files are the most suitable models to exchange with the outside in case character type is not the base of the exchange.

As in the previous case, also in these circumstances input control is feasible through form string modes.

This implies to terminate these requests only when bytes number collected by the device is the same as the size of instantiated type binary representation.

Omitting identification of page terminator can be penalized when teletype characterization is used to handle serial communication lines connected to host computers. In these cases implementation allows to restore in SKIP\_LINE page terminator recognition through FORM string LINE\_EDIT qualifier (see Selection of external files attributes).

### 13.2 Devices Handled By Real Time Protocol

Not supported

### 13.3 Basic Types

In the STANDARD package, the following types are defined:

- INTEGER = on 16 bits, its range is -32768..32767
- LONG\_INTEGER = on 32 bits, its range is -2147483648..2147483647

From the point of view of the admitted values range, to these types respectively correspond - in Ada/Digital - the types:

- . SHORT\_INTEGER
- . INTEGER

For this reason, it would be better not to use standard definitions but to define some new types which are made to correspond to equal representations in the two implementations; what said can be extended to NATURAL and POSITIVE subtypes.

Predefined types FLOAT and LONG\_FLOAT are represented respectively with 6 digits on 32 bits and with 15 digits on 64 bits.

This representation coincides to the ADA/Digital one for the types with the same name.

To obtain the transportability on MARA, in Ada/Digital you must not use:

- the LONG\_LONG\_FLOAT type (33 digits on 128 bits) which has not an equivalent in our compiler
- the LONG\_FLOAT pragma which originates a different representation for LONG\_FLOAT objects (15 digits on 64 bits)

The generic FLOAT\_MATH\_LIB ([MATH]) package which is present at the first level of Mara program library, can be implemented with FLOAT, LONG\_FLOAT types or with types derived from them.

In conclusion, to assure the transportability between these two systems, it is advisable to define and use the following BASIC\_TYPES package:

package BASIC\_TYPES is

```

    type INTEGER_16      is new INTEGER;                - - For
DDC
    type INTEGER_32 is new LONG_INTEGER;                - - For DDC
- -   type INTEGER_16      is new SHORT_INTEGER;        - - For DEC
- -   type INTEGER_32      is new INTEGER;              - - For DEC

    subtype NATURAL_16    is INTEGER_16 range 0..INTEGER_16LAST;
    subtype NATURAL_32    is INTEGER_32 range 0..INTEGER_32LAST;

    subtype POSITIVE_16   is INTEGER_16 range 1..INTEGER_16LAST;
    subtype POSITIVE_32   is INTEGER_32 range 1..INTEGER_32LAST;

    type FLOAT_32         is new FLOAT;                 - - For
DEC & DDC
    type FLOAT_64         is new LONG_FLOAT;

```

end BASIC\_TYPES;

However, it is recommended to do no assumption on how a certain type (particularly if complex) of objects are memory represented, and, if needed, to resort to explicit representation clauses.

#### 13.4 Ada/DDC programs sementing.

An Ada/DDC program may have a certain number of associated data-segments, each if them corresponding to a program-library level. Objects declared in a library package or in a package that, in any case, is not contained in a subprogram declaration (but in another package) and which are at that library level take up room in a data-segment of a given level.

The maximum dimension of each data-segment is 64 k-bytes; this value imposes a limit to the quantity and to the dimension of the objects which are allocated in it.

We define as library tasks the tasks whose declaration (in case of the only type) immediately appears inside a library package, or inside a package which is directly inside a library package and so on; after a point of this type is defined as a 0 level point.

To each of these tasks, a memory segment is dynamically associated; in this segment, this task stack is allocated together with the stacks of all the others tasks depending on it ([LRM 83], par. 9.4).

The main-task handling is similar to that of library tasks.

The dimension of the memory segment associated to the main-task is fixed to the moment of the program generation by means of the stacksegment size SCL directive ([SCL]). On the contrary, for a library task it can be fixed by means of the SET\_CHILD\_SEGMENT\_SIZE function recall ([APX A - ADA\_RTS]);

In case of lack of this one, the dimension of the task segment processing its declaration, or performing its dynamic allocation is used (through new).

The stack dimension of a task can be fixed by means of a length clause ([LRM 83], par. 13.2); and the main-task one through the option provided by the linker ([CLU], par. 3).

Obviously, the dimension of a segment associated to a library task must be large enough to contain the stack of this task and those of the tasks depending on it. If in the segment there is no room for the stack of the nth task, the TASKING-ERROR exception is raised ([LRM 83], par. 9.3).

Also for each of these segments, the maximum dimension is 64 k-bytes.

As tasks handling in Ada/Digital is completely different, this exception handling is recommended in order to easily detect the cause of different behaviours in applications brought from VAX to Mara.

As to the code, the dimension of a single compilation unit segment cannot exceed 32 k-bytes; see also, ([Users guide] par. 6.1.1) what about the clusterization of the code coming from different compilation units.

It must be taken into account that Mara Software Factory tools imply that a program occupies a maximum of 255 memory segments, code and data included; thus, in case of Ada programs, the segments containing the tasks stacks must not be counted.

Finally, see ([Users guide] par. 9.6) what about the way in which the compiler associates declarative items to different memory areas.

### 13.5 Shared variables.

Ada permits the use of shared variables, i.e. variables simultaneously accessible to various tasks. Note that the variables declared in a package body are not visible but they can be still shared by package procedures; thus, an indirect access to them is possible by various

tasks.

A compiler is allowed to make the two following assumptions ([LRM 83], par. 9.11):

- if between two synchronization points a task reads a scalar or access type shared variable, then the variable is modified by no other task in a moment between these two points.
- if between two synchronization points a task modifies a scalar or access type shared variable, then the variable is either read or modified by no other task in a moment between these two points.

A program violating the previous assumptions is formally wrong. May be that its execution is correct, but it isn't any more if the compiler or simply the optimization is changed.

In fact, their application is permitted also when shared variables are involved; thus, a compiler can maintain their value in a register instead of updating it continuously in memory. The only guarantee that the compiler must provide is that the variable is actually updated when a synchronization point is reached, and that after this point a copy - for instance contained in a register - is not used.

No assumption can be done by a programmer on not mentioned type shared variables handling. In particular, on `FILE_TYPE` objects (which are limited private), two tasks can have no consistency guarantee. In case of sharing, a solution can be that of defining a task which - being the only authorized one - accesses to I/O services and exports their specification with as many tasks entries.

For a compact management of the problem, in case of `SEQUENTIAL_IO`, the `SYNCHRONIZED_SEQUENTIAL_IO` package reported below can be used as a trace; the same solution can be adopted for the generic `DIRECT_IO` package and for the non-generic `TEXT_IO` one.

```
with sequential_io; use sequential_io;
with io_exceptions;
```

```
generic
type element_type is private;
```

```
package synchronized_sequential_io
```

```
    procedure create (    file      :    in out file_type;
                          mode      :    in file_mode   := out_file;
                          name      :    string         :=    ;
                          form      :    string         :=    );
```

```
    procedure read      ( file      :    in    file_type;
                          item      :    out  element_type);
private
```

```
end synchronized_sequential_io;
```

```
package body synchronized_sequential_io is
```

task synchronize is

```
entry create (   file      :   in out file_type;
                mode      :   in file_mode   := out_file;
                name      :   string         :=   ;
                form      :   string         :=   );
```

```
entry read ( file      :   in   file_type;
             item      :   out  element_type);
```

end synchronize;

```
procedure create (   file      :   in out file_type;
                   mode      :   in file_mode   := out_file;
                   name      :   string         :=   ;
                   form      :   string         :=   ) is
```

```
begin
    synchronize.create(file,mode,name,form);
end;
```

```
procedure read (   file      :   in   file_type;
                 item      :   out  element_type) is
```

```
begin
    synchronize.read(file,item);
end;
```

task body synchronize is

```
begin
    select
        accept create (   file      :   in out file_type;
                        mode      :   in file_mode   := out_file;
                        name      :   string         :=   ;
                        form      :   string         :=   )
            do
                sequential_io.create(file,mode,name,form);
            end;
```

```
        or
        accept read (   file      :   in   file_type;
                      item      :   out  element_type)
            do
                sequential_io.read(file,item);
            end;
```

```
        or
        terminate;
```

end select;

end synchronize;

### 13.6 Elaboration Order



The elaboration order of the library units needed by a main program, is the one specified in ([LRM 83] par. 10.5).

About the matter it must be pointed out that no assumption can be done on the elaboration order of two packages bodies, also when one of them is a with of the other. For instance:

```
package A is
function F return INTEGER;
end A;
```

```
package B is
procedure START;
end B;
```

```
with A;
package body B is
III: INTEGER:= A.F;
procedure START is
begin
    null;
end START;
end B;
```

In this example the elaboration order can be wrong. In similar cases, the use of the ELABORATE pragma is recommended ([LRM 83], par. 10.5).

### 13.7 Tasking

It is recommended not to use the PRIORITY pragma in order to accomplish the synchronization among tasks, and to refer to the knowledge of scheduling mechanisms only within the limits of what is specified in ([LRM 83], par. 9.8).

### 13.8 I/O Interrupt.

The handling of Mara interrupts coming from interface modules (HIM) must be implemented by using the mechanisms offered by the operating system.

Thus, it is necessary to recall the primitives provided by ([Mara286\_2], cap. 11) in order to:

- create a virtual line
- connect the line to a HIM
- suspend an Ada task waiting for the generation of an interrupt coming from that HIM

## 14 Installation Guide

Purpose of the chapter is to provide Digital VAX/VMS System Manager with an installation procedure for DDC Ada 8086/80286 Compiler System and Alenia Software Components.

The whole Ada installation kit is composed of the following products:

- DACS86 (Ada compiler and Linker)
- ADARTS (Ada run time support)
- ADABIO (Ada BASIC I/O)
- IDA (Cross Debugger)

The installation of these products implies:

- Delivery tape is down loaded
- Mara Factory has already been installed (Kernel, SCL, GENRP...)

#### 14.1 Installation Procedure

The operation preliminary to installation are:

- To login (from system to user)
- To make sure that 30.000 memory block are available
- To make sure that the logics (such as KER286A0, KER286A1, GATELBA, etc) are correctly assigned;
- To make sure that hosted Intel Software Factory v. 3.2 and following are installed
- To make sure that version /ISIS of INTEL ASM286 and BND286 products are available;
- To create a directory for ADARTS products and associate ADARTSA0 and ADARTSA1 logics name to this area
- to create a directory for ADABIO products and associate ADABIOA0 and ADABIOA1 logic names to this area
- As to IDA installation refer to [IDA286] document
- To create a directory for DACS86 products and associate DACS86A0 logic name to this area

Ada command verb can be chosen as you like provided that the first 4 letters are different from any other DCL command.

Installation procedure requires System Identification Register (SID), obtained through function f\$getsyi("sid") and relative check sums supplied by Alenia in a letter enclosed to delivery tape.

Installation command file requires, as input parameter, a file containing information relative to license.

The structure of License Check File (<file>.CKS) is:

```
<expiration date>
<serial number>
{<sid>
<check sum1>
<check sum2>}
```

<blank line>  
[<compiler name>]

{...} stays for iteration, while [...] stays for optionality. At least one <sid> of a VAX machine must be indicated.

The <compiler name> field indicates the name used to indicate the compiler tool.

The default name is PM286.

The recommended name is ADA286.

For the installation command see the document "Installation Guide" of the product DACS86.

The installation procedure also executes an ADA program which tests the correctness of the installation just performed and the correctness of all tools installed.

To generate the test program an Ada library and a default graph are created.

The result of program generation is a file called HELLO.LTL.

After installation, each user must execute

@DACs86A0:ADAUSE.COM

to have compiler system available for login current session.

## 14.2 Installation products

The following files are produced when the installation is well terminated:

ADA286.SCL

- ADA286.GRA
- HELLO.LTL
- MULTii.LTL

The products list follows which must be present in any tape configuration including DACS86.

For a consistent installation, products order in the list is important, too.

- 1) LICENSE from 00.00
- 2) SCL286 from 04.0x
- 3) GENRP from 04.0x
- 4) KER286 from 04.00
- 5) ADARTS from 00.00

6) ADABIO      from 00.00

7) DACS86      from 04.3x

Note: To run Installation Procedure successfully, System Manager must be sure that Intel Software Factory version v3.2 or subsequent, is installed in VAX/VMS (see [INT]);

## 15      APPENDIX A [APX A]

### 15.1    BASIC\_IO

```
Basic_io product
generic
    form : string := NODOUBLE_BUFFER &
    ,LINE_FOLDED &
    ,NOLINE_EDIT &
    ,NOSHARE_IN &
    ,NOSHARE_OUT &
    ,NOSHARE_INOUT &
    ,ECHO &
    ,TERMINATOR &
    ,TIMED ;
    ci_form : string :=      SHARE_OUT &
    ,LINE_EDIT ;
    co_form : string :=      NOTIMED ;
package basic_io is end;
```

### 15.2    BASIC\_IO\_CONFIGURATION

Provide for basic\_io configuration at program level.  
with calendar;  
generic

```
    buffer_size            : integer;
    time_out               : calendar.day_duration;
    number_of_files        : natural;
package basic_io_configuration is end;
```

### 15.3    DEFAULT\_FORM

```
generic
value : string;
package default_form is
end;
```

### 15.4    DEFAULT\_DEVICES

```

generic
input_name      : string;
output_name     : string;
input_form      : string;
output_form     : string;
package default_devices is
end;

```

## 15.5 ADA\_RTS

```

with system;
package ADA_RTS is

```

- Tasking support issues

```

type computer_id is range 0 .. 15;

```

```

type PSA_REC is

```

```

  record
    Unit_no           : system.unsignedword;
    Exception_id      : system.unsignedword;
    Mara_code         : system.unsignedword;
    Exception_offset   : system.unsignedword;
    Exception_selector : system.segmentid;
  end record;

```

```

function MY_COMPUTER return computer_id;
function SET_CHILD_COMPUTER
  (computer : in computer_id)
  return computer_id;

```

```

function MY_STACK_SIZE return long_integer;

```

```

function SET_CHILD_SEGMENT_SIZE
  (stack_size : in long_integer)
  return long_integer;

```

-- Exception support issues

```

function EXCEPTION_CODE return system.unsignedword;

```

```

-- Set the tracer for unhandled exception
-- as the exception handler of the current function

```

```

procedure unhandled_exception_tracer;

```

```

private

```

```

pragma interface (ASM86, MY_COMPUTER);
pragma interface_spelling (MY_COMPUTER, "E_Computer");

```

```

pragma interface (ASM86, SET_CHILD_COMPUTER);
pragma interface_spelling (SET_CHILD_COMPUTER, "E_Child_Computer");

```

```

pragma interface (ASM86, MY_STACK_SIZE);
pragma interface_SPELLING (MY_STACK_SIZE, "E_Stack_Size")

pragma interface (ASM86, SET_CHILD_SEGMENT_SIZE);
interface_spelling (SET_CHILD_SEGMENT_SIZE, "E_Child_Segment_Size");

pragma interface (ASM86, EXCEPTION_CODE);
pragma interface_spelling (EXCEPTION_CODE, "E_Error_Code");

pragma interface (ASM286, TRACE_INFO);

pragma interface (ASM286, GetExceptionId);
pragma interface_spelling (RTS_GetExceptionId,
    "R1EHGE?GetExceptionId");

pragma interface (ASM86, RTS_GetExceptionSpelling);
pragma interface_spelling (RTS_GetExceptionSpelling,
    "R1EHGE?GetExceptionSpelling")

end ADA_RTS;

```

## 15.6 REAL\_TIME\_CLOCK

```

package REAL_TIME_CLOCK is

type TIME is range 0..1000*60*60*24 - 1;  -- ms in a day

procedure SETTIME (TIME_VALUE : in TIME);

function GETTIME return TIME;

package TIME_IO is
procedure GET( item : out TIME);
procedure PUT( item : in TIME);
end TIME_IO;

private

pragma INTERFACE (PLM86, SETTIME);
pragma INTERFACE (PLM86, GETTIME);

end REAL_TIME_CLOCK;

```

## 15.7 BASIC\_TYPES

```

package BASIC_TYPES is

```

This package is intended to help Ada programs to be portable respect to the word size of target machines. Programmers should use these types with the explicit size indication and refrain from using INTEGER, NATURAL, POSITIVE and FLOAT standard types.

```

type INTEGER_16 is new INTEGER;  -- for DDC

```

```

type INTEGER_32 is new LONG_INTEGER;      -- for DDC

- type INTEGER_16 is new SHORT_INTEGER;   -- for DEC
- type INTEGER_32 is new INTEGER;         -- for DEC

subtype NATURAL_16 is INTEGER_16 range 0 .. INTEGER_16LAST;
subtype NATURAL_32 is INTEGER_32 range 0 .. INTEGER_32LAST;

subtype POSITIVE_16 is INTEGER_16 range 1 .. INTEGER_16LAST;
subtype POSITIVE_32 is INTEGER_32 range 1 .. INTEGER_32LAST;

type FLOAT_32 is new FLOAT;                -- For DDC and DEC
type FLOAT_64 is new LONG_FLOAT;           -- For DDC and DEC

end BASIC_TYPES;

```

### 15.8 ADDRESS\_IMAGE

```

- 21_FEB_1989 12:21:00.77 /MARTIN
with SYSTEM; use SYSTEM;
with UNCHECKED_CONVERSION;
with BASIC_TYPES; use BASIC_TYPES;
function ADDRESS_IMAGE ( V : in ADDRESS )
return STRING is

- This function is suitable for VAX targets with DEC or DDC Ada

function CONV is new UNCHECKED_CONVERSION
  (SOURCE => ADDRESS,
   TARGET => INTEGER_32 );

begin

return INTEGER_32' IMAGE ( CONV( V ) );

end ADDRESS_IMAGE;

```

### 15.9 CALENDAR\_IMAGE

```

with CALENDAR; use CALENDAR;
package CALENDAR_IMAGE is

function DATE_IMAGE( T : in TIME ) return STRING;

function DATE_IMAGE return STRING;

function TIME_IMAGE( T : in TIME ) return STRING;

function TIME_IMAGE return STRING;

end CALENDAR_IMAGE;

```

## 15.10 GENERIC\_DUMPS

```
- 1-DEC-1988 16:15:09.22 /MARTIN
with TEXT_IO;           use TEXT_IO;
with CALENDAR;          use CALENDAR;
with BASIC_TYPES;       use BASIC_TYPES;
package GENERIC_DUMPS is
```

```
type BOX is (UNBOXED, BOXED);
```

```
DUMP_FILE : FILE_TYPE;
```

The following procedures must be called at the start and at the end of a test run.

```
procedure START_RUN( UNIT_NAME : in STRING );
```

```
procedure END_RUN;
```

The following procedure allows labelling the start of the dump output relative to a certain test. These labels can be used to facilitate correlation between a program and the dump output it produced or comparison between the dumps produced in different executions.

```
procedure START-TEST ( TEST_NUMBER : in POSITIVE_16;
                      DESCRIPTION : in STRING := '' );
```

The following procedures help the user to generate dump procedures for his own record types.

```
procedure START_ARRAY      ( NAME : in STRING; B : in BOX );
```

```
procedure START_RECORD     ( NAME : in STRING; B : in BOX );
```

```
procedure END_ARRAY        ( B: in BOX);
```

```
procedure END_RECORD       ( B : in BOX);
```

Basic procedure for dumping string literals

```
procedure DUMP      ( S : in STRING;
                    B : in BOX := UNBOXED );
```

Basic procedure for dumping objects of type STRING

```
procedure DUMP      ( NAME : in STRING;
                    V      : in STRING;
                    B      : in BOX := UNBOXED );
```

Procedure for dumping objects of type CALENDAR.TIME



```

procedure DUMP      ( NAME : in STRING;
                     V      : in CALENDAR.TIME;
                     B      : in BOX := UNBOXED );

```

The following generics enable the user to generate dump procedures for his own simple types and array types.

```

generic
type DISCRETE_TYPE is (<>);
procedure DISCRETE_DUMP      ( NAME : in STRING;
                              V      : in DISCRETE_TYPE;
                              B      : in BOX := UNBOXED );

```

```

generic
type FIXED_TYPE is delta <>;
procedure FIXED_DUMP      ( NAME : in STRING;
                           V      : in FIXED_TYPE;
                           B      : in BOX := UNBOXED );

```

```

generic
type FLOAT_TYPE is digits <>;
procedure FLOAT_DUMP      ( NAME : in STRING;
                           V      : in FLOAT_TYPE;
                           B      : in BOX := UNBOXED );

```

```

generic
type ACCESS_TYPE is private;
procedure ACCESS_DUMP      ( NAME : in STRING;
                            V      : in ACCESS_TYPE;
                            B      : in BOX := UNBOXED );

```

```

generic
type INDEX_TYPE is (<>);
type COMPONENT_TYPE is private;
type ARRAY_TYPE is array( INDEX_TYPE ) of COMPONENT_TYPE;
with procedure DUMP      ( NAME : in STRING;
                          V      : in COMPONENT_TYPE;
                          B      : in BOX := UNBOXED ) is <>;
procedure ARRAY_DUMP      ( NAME : in STRING;
                          V      : in ARRAY_TYPE;
                          B      : in BOX := UNBOXED );

```

```

generic
type INDEX1_TYPE is (<>);
type INDEX2_TYPE is (<>);
type COMPONENT_type is private;
type ARRAY_TYPE is array( INDEX1_TYPE, INDEX2_TYPE )
of COMPONENT_TYPE;
with procedure DUMP      ( NAME : in STRING;
                          V      : in COMPONENT_TYPE;
                          B      : in BOX := UNBOXED ) is <>;
procedure ARRAY2_DUMP      ( NAME : in STRING;
                          V      : in ARRAY_TYPE;
                          B      : in BOX := UNBOXED );
end GENERIC_DUMPS;

```

## 15.11 BASIC\_DUMPS

```
- 1-DEC-1988 16:17:04.65 /MARTIN
with BASIC_TYPES; use BASIC_TYPES;
with GENERIC_DUMPS use GENERIC_DUMPS;
package BASIC_DUMPS is
```

These are instantiations, for the predefined types and the types defined in package BASIC\_TYPES, of the generic dumping procedures in GENERIC\_DUMPS.

```
procedure DUMP is new DISCRETE_DUMP
( DISCRETE_TYPE => INTEGER_16 );
procedure DUMP is new DISCRETE_DUMP
( DISCRETE_TYPE => INTEGER_32 );
```

```
procedure DUMP is new DISCRETE_DUMP
( DISCRETE_TYPE => BOOLEAN );
```

```
procedure DUMP is new DISCRETE_DUMP
( DISCRETE_TYPE => CHARACTER );
```

```
procedure DUMP is new FIXED_DUMP
( FIXED_TYPE => DURATION );
```

```
procedure DUMP is new FLOAT_DUMP
( FLOAT_TYPE => FLOAT_32 );
```

```
procedure DUMP is new FLOAT_DUMP
( FLOAT_TYPE => FLOAT_64 );
```

```
end BASIC_DUMPS;
```

## 15.12 RANDOM

```
- 27-FEB-1989 08:32:13.50 /MARTIN
with BASIC_TYPES; use BASIC_TYPES;
package RANDOM is
```

The algorithm is taken from the article:

Random number generators: good ones are hard to find  
S.K. Park and K.W. Miller  
Communications of the ACM, Oct 1988, Vol 31, Number 10.

The user may set SEED to any NATURAL\_32 value

```
SEED : NATURAL_32 := 1;
```

```
function RAND return NATURAL_32; -- 0 .. 2,147,483,647
function RAND return FLOAT_32; -- 0.0 .. 1.0
```

```
function RAND return INTEGER_16;  -- 32768 .. 32767

end RANDOM;
```

### 15.13 DUMP 287

- Alenia
- Informatics Factory
- August 1989

Procedure to dump 287 stack top trace buffer contents. The procedure is for use in debugging of coprocessor algorithms written in ASM286. In particular it is used for the debugging of the math library MATHLIB .

```
with GENERIC_DUMPS; use GENERIC_DUMPS;
procedure DUMP287 ( B : in BOX := UNBOXED ) is

    type REF_LABEL is access INTEGER_16;

    function GET_NREC return NATURAL_16;
    pragma INTERFACE (PLM_NOAC, GET_NREC);
    pragma INTERFACE_SPELLING ( GET_NREC, "mathlib_GET_NREC" );

    function GET_SAVED_LABEL ( N: in POSITIVE_16) return REF_LABEL;
    pragma INTERFACE ( PLM_NOACF, GET_SAVED_LABEL );
    pragma INTERFACE ( GET_SAVED_LABEL, "mathlib_GET_SAVED_LABEL");

    function GET_SAVED_ST ( N: in POSITIVE_16) return FLOAT_64;
    pragma INTERFACE ( PLM_NOACF, GET_SAVED_ST );
    pragma INTERFACE ( GET_SAVED_LABEL, "mathlib_GET_SAVED_ST");

    procedure DUMP is new ACCESS_DUMP( REF_LABEL );

    NREC: NATURAL_16;

    begin

        NREC := GET_NREC;
        START_ARRAY( "287/387 STACK TOP TRACES", 8);
        for I in 1..NREC loop
            START_RECORD ("TRACE_RECORD", UNBOXED);
            DUMP( "TRACE_LABEL", GET_SAVED_LABEL(I));
            DUMP( "ST_VALUE", GET_SAVED_ST(I));
        end loop;
        END_ARRAY(b);

    END DUMP287;
```

### 15.14 MATH\_LIB

- Alenia
- Informatics Factory
- August 1989

--This file contains the specifications of packages MATH\_LIB\_32  
 --and MATHLIB\_64 for use on Mara286 with MON286, MUL286, or MUL386  
 --boards with or without their respective coprocessor chips.  
 --The body is contained in an ASM286 module called MATHLIB.

with BASIC\_TYPES; use BASIC\_TYPES;  
 package MATH\_LIB\_32 is

```

function SQRT      ( X : in FLOAT_32 )      return FLOAT_32;
function LOG       ( X : in FLOAT_32 )      return FLOAT_32;
function LOG10     ( X : in FLOAT_32 )      return FLOAT_32;
function EXP       ( X : in FLOAT_32 )      return FLOAT_32;
function Y2X       ( Y, X : in FLOAT_32 )   return FLOAT_32;
function Y2I       ( Y : in FLOAT_32;
                    I  : in INTEGER_16 )     return FLOAT_32;
function SIN       ( X : in FLOAT_32 )      return FLOAT_32;
function COS       ( X : in FLOAT_32 )      return FLOAT_32;
function TAN       ( X : in FLOAT_32 )      return FLOAT_32;
function SIN_SC    ( X : in FLOAT_32 )      return FLOAT_32;
function COS_SC    ( X : in FLOAT_32 )      return FLOAT_32;
function ATAN2     ( Y, X : in FLOAT_32 )   return FLOAT_32;
function ATAN      ( Y : in FLOAT_32 )      return FLOAT_32;
function ASIN      ( X : in FLOAT_32 )      return FLOAT_32;
function ACOS      ( X : in FLOAT_32 )      return FLOAT_32;
function SINH      ( X : in FLOAT_32 )      return FLOAT_32;
function COSH      ( X : in FLOAT_32 )      return FLOAT_32;
function TANH      ( X : in FLOAT_32 )      return FLOAT_32;

```

--Out of range arguments cause exception NUMERIC\_ERROR to be raised.

private

```

pragma INTERFACE( C_REVERSE_NOACF, SQRT );
pragma INTERFACE_SPELLING( LOG, "mathlib_SQRT" );

pragma INTERFACE( C_REVERSE_NOACF, LOG );
pragma INTERFACE_SPELLING( LOG, "mathlib_LOG" );

pragma INTERFACE( C_REVERSE_NOACF, LOG10 );
pragma INTERFACE_SPELLING( LOG10, "mathlib_LOG10" );

pragma INTERFACE( C_REVERSE_NOACF, EXP );
pragma INTERFACE_SPELLING( EXP, "mathlib_EXP" );

pragma INTERFACE( C_REVERSE_NOACF, Y2X );
pragma INTERFACE_SPELLING( Y2X, "mathlib_Y2X" );

pragma INTERFACE( C_REVERSE_NOACF, Y2I );
pragma INTERFACE_SPELLING( Y2I, "mathlib_Y2I" );

pragma INTERFACE( C_REVERSE_NOACF, SIN );
pragma INTERFACE_SPELLING( SIN, "mathlib_SIN" );

```

```

pragma INTERFACE( C_REVERSE_NOACF, COS );
pragma INTERFACE_SPELLING( COS, "mathlib_COS" );

pragma INTERFACE( C_REVERSE_NOACF, TAN );
pragma INTERFACE_SPELLING( TAN, "mathlib_TAN" );

pragma INTERFACE( C_REVERSE_NOACF, COS_SC );
pragma INTERFACE_SPELLING( COS_SC, "mathlib_COS_SC" );

pragma INTERFACE( C_REVERSE_NOACF, ASIN );
pragma INTERFACE_SPELLING( ASIN, "mathlib_ASIN" );

pragma INTERFACE( C_REVERSE_NOACF, ACOS );
pragma INTERFACE_SPELLING( ACOS, "mathlib_ACOS" );

pragma INTERFACE( C_REVERSE_NOACF, ATAN );
pragma INTERFACE_SPELLING( ATAN, "mathlib_ATAN" );

pragma INTERFACE( C_REVERSE_NOACF, ATAN2 );
pragma INTERFACE_SPELLING( ATAN2, "mathlib_ATAN2" );

pragma INTERFACE( C_REVERSE_NOACF, SINH );
pragma INTERFACE_SPELLING( SINH, "mathlib_SINH" );

pragma INTERFACE( C_REVERSE_NOACF, COSH );
pragma INTERFACE_SPELLING( COSH, "mathlib_COSH" );

pragma INTERFACE( C_REVERSE_NOACF, TANH );
pragma INTERFACE_SPELLING( TANH, "mathlib_TANH" );

end MATH_LIB_32;

```

- Alenia  
- Informatics Factory  
- August 1989

This file contains the specification of package MATH\_LIB\_64 for use on Mara286 with the Intel CEL287 library.

with BASIC\_TYPES; use BASIC\_TYPES;  
package MATH\_LIB\_64 is

```

function SQRT      ( X : in FLOAT_64 )      return FLOAT_64;
function LOG       ( X : in FLOAT_64 )      return FLOAT_64;
function LOG10     ( X : in FLOAT_64 )      return FLOAT_64;
function EXP       ( X : in FLOAT_64 )      return FLOAT_64;
function Y2X       ( Y, X : in FLOAT_64 )   return FLOAT_64;
function Y2I       ( Y : in FLOAT_64;
                    I : in INTEGER_16 )      return FLOAT_64;
function SIN       ( X : in FLOAT_64 )      return FLOAT_64;
function COS       ( X : in FLOAT_64 )      return FLOAT_64;
function TAN       ( X : in FLOAT_64 )      return FLOAT_64;
function SIN_SC    ( X : in FLOAT_64 )      return FLOAT_64;
function COS_SC    ( X : in FLOAT_64 )      return FLOAT_64;

```

```

function ATAN2      ( Y, X : in FLOAT_64 )      return FLOAT_64;
function ATAN      ( Y : in FLOAT_64 )          return FLOAT_64;
function ASIN      ( X : in FLOAT_64 )          return FLOAT_64;
function ACOS      ( X : in FLOAT_64 )          return FLOAT_64;
function SINH      ( X : in FLOAT_64 )          return FLOAT_64;
function COSH      ( X : in FLOAT_64 )          return FLOAT_64;
function TANH      ( X : in FLOAT_64 )          return FLOAT_64;

```

Out of range arguments cause exception `NUMERIC_ERROR` to be raised.

private

The use of the language `C_REVERSE_NOACF` in the following `INTERFACE` pragmas is to work around a bug in the code generator. This bug has been fixed in a version not yet released.

```

pragma INTERFACE( C_REVERSE_NOACF, SQRT );
pragma INTERFACE_SPELLING( SQRT, "mathlib_SQRT" );

pragma INTERFACE( C_REVERSE_NOACF, LOG );
pragma INTERFACE_SPELLING( LOG, "mathlib_LOG" );

pragma INTERFACE( C_REVERSE_NOACF, LOG10 );
pragma INTERFACE_SPELLING( LOG10, " mathlib_log10" );

pragma INTERFACE( C_REVERSE_NOACF, EXP );
pragma INTERFACE_SPELLING( EXP, "mathlib_EXP" );

pragma INTERFACE( C_REVERSE_NOACF, Y2X );
pragma INTERFACE_SPELLING( Y2X, "mathlib_Y2X " );

pragma INTERFACE( C_REVERSE_NOACF, Y2I );
pragma INTERFACE_SPELLING( Y2I, "mathlib_Y2I" );

pragma INTERFACE C_REVERSE_NOACF, SIN );
pragma INTERFACE_SPELLING( SIN, "mathlib_SIN " );

pragma INTERFACE( C_REVERSE_NOACF, COS );
pragma INTERFACE_SPELLING( COS, "mathlib_COS" );

pragma INTERFACE( C_REVERSE_NOACF, TAN );
pragma INTERFACE_SPELLING( TAN, "mathlib_TAN" );

pragma INTERFACE( C_REVERSE_NOACF, COS_SC );
pragma INTERFACE_SPELLING( COS_SC, "mathlib_COS_SC" );

pragma INTERFACE( C_REVERSE_NOACF, SIN_SC );
pragma INTERFACE_SPELLING( SIN_SC, "mathlib_SIN_SC" );

pragma INTERFACE( C_REVERSE_NOACF, ASIN );
pragma INTERFACE_SPELLING( ASIN, " mathlib_ASIN" );

pragma INTERFACE( C_REVERSE_NOACF, ACOS );
pragma INTERFACE_SPELLING( ACOS, "mathlib_ACOS" );

pragma INTERFACE( C_REVERSE_NOACF, ATAN );

```

```

pragma INTERFACE_SPELLING( ATAN, " mathlib_ATAN" );

pragma INTERFACE( C REVERSE NOACF, ATAN2 );
pragma INTERFACE_SPELLING( ATAN2, " mathlib_ATAN2" );

pragma INTERFACE( C REVERSE NOACF, SINH );
pragma INTERFACE_SPELLING( SINH, " mathlib_SINH " );

pragma INTERFACE( C REVERSE NOACF, COSH );
pragma INTERFACE_SPELLING( COSH, " mathlib_COSH " );

pragma INTERFACE( C REVERSE NOACF, TANH );
pragma INTERFACE_SPELLING( TANH, " mathlib_TANH" );

end MATH_LIB_64;

```

## 16 APPENDIX B [APX B]

### 16.1 ADA286.GRA

```
system ADA286;
```

```
program template ADA286 large
```

```
code PRIVATE
```

```
data PRIVATE NODAL;
```

module	:FMS:ADARTSA1/RTSDATA.OBJ	relocatable;
module	:FMS:ADABIOA1/BIODATA.OBJ	relocatable;
module	ADA286	relocatable;
module	:FMS:DACS86A0/ROOT.LIB	relocatable;
module	:FMS:DACS86A0/RTHELP286.LIB	relocatable;
module	:FMS:KER286A0/ADAUS.LIB	relocatable;
module	:FMS:ADABIOA1/BINDER286.LIB	relocatable;
module	:FMS:ADARTSA1/BINDER286.LIB	relocatable;
module	:FMS:KER286A0/ADAUS.LIB	relocatable;
module	GATELBA	relocatable;

```
initial procedure cg_adamainprogram
stacksegment size = 0FF00H;
```

```
end program;
```

```
$ eject
```

```
program template MULTI_ADA286 large
```

```
code PRIVATE
```

```
data PRIVATE NODAL;
```

```
subprogram GENERAL repeatable;
```

module	:FMS:ADARTSA1/RTSDATA.OBJ	relocatable;
module	:FMS:ADABIOA1/BIODATA.OBJ	relocatable;
module	ELABORATION source asm;	
\$include(Repeated)		
module	:FMS:DACS86A0/ROOT.LIB	relocatable;
module	:FMS:DACS86A0/RTHELP286.LIB	relocatable;
module	:FMS:KER286A0/ADAUS.LIB	relocatable;

```

module      :FMS:ADABIOA1/BINDER286.LIB      relocatable;
module      :FMS:ADARTSA1/BINDER286.LIB      relocatable;
module      :FMS:KER286A0/ADAUS.LIB           relocatable;
module      GATELBA                           relocatable;
end subprogram;

subprogram ZONE_0 optional;
$include(Zone0)
module      :FMS:DACS86A0/ROOT.LIB            relocatable;
module      :FMS:DACS86A0/RTHELP286.LIB       relocatable;
module      :FMS:KER286A0/ADAUS.LIB           relocatable;
module      :FMS:ADABIOA1/BINDER286.LIB       relocatable;
module      :FMS:ADARTSA1/BINDER286.LIB       relocatable;
module      :FMS:KER286A0/ADAUS.LIB           relocatable;
module      GATELBA                           relocatable;
end subprogram;

subprogram ZONE_1 optional;
$include(Zone1)
module      :FMS:DACS86A0/ROOT.LIB            relocatable;
module      :FMS:DACS86A0/RTHELP286.LIB       relocatable;
module      :FMS:KER286A0/ADAUS.LIB           relocatable;
module      :FMS:ADABIOA1/BINDER286.LIB       relocatable;
module      :FMS:ADARTSA1/BINDER286.LIB       relocatable;
module      :FMS:KER286A0/ADAUS.LIB           relocatable;
module      GATELBA                           relocatable;
end subprogram;

subprogram ZONE_2 optional;
$include(Zone2)
module      :FMS:DACS86A0/ROOT.LIB            relocatable;
module      :FMS:DACS86A0/RTHELP286.LIB       relocatable;
module      :FMS:KER286A0/ADAUS.LIB           relocatable;
module      :FMS:ADABIOA1/BINDER286.LIB       relocatable;
module      :FMS:ADARTSA1/BINDER286.LIB       relocatable;
module      :FMS:KER286A0/ADAUS.LIB           relocatable;
module      GATELBA                           relocatable;
end subprogram;

subprogram ZONE_3 optional;
$include(Zone3)
module      :FMS:DACS86A0/ROOT.LIB            relocatable;
module      :FMS:DACS86A0/RTHELP286.LIB       relocatable;
module      :FMS:KER286A0/ADAUS.LIB           relocatable;
module      :FMS:ADABIOA1/BINDER286.LIB       relocatable;
module      :FMS:ADARTSA1/BINDER286.LIB       relocatable;
module      :FMS:KER286A0/ADAUS.LIB           relocatable;
module      GATELBA                           relocatable;
end subprogram;

subprogram ZONE_4 optional;
$include(Zone4)
module      :FMS:DACS86A0/ROOT.LIB            relocatable;
module      :FMS:DACS86A0/RTHELP286.LIB       relocatable;
module      :FMS:KER286A0/ADAUS.LIB           relocatable;
module      :FMS:ADABIOA1/BINDER286.LIB       relocatable;

```



```

module      :FMS:ADARTSA1/BINDER286.LIB      relocatable;
module      :FMS:KER286A0/ADAUS.LIB          relocatable;
module      GATELBA                          relocatable;
end subprogram;

```

subprogram ZONE\_5 optional;

\$include(Zone5)

```

module      :FMS:DACS86A0/ROOT.LIB           relocatable;
module      :FMS:DACS86A0/RTHELP286.LIB      relocatable;
module      :FMS:KER286A0/ADAUS.LIB          relocatable;
module      :FMS:ADABIOA1/BINDER286.LIB      relocatable;
module      :FMS:ADARTSA1/BINDER286.LIB      relocatable;
module      :FMS:KER286A0/ADAUS.LIB          relocatable;
module      GATELBA                          relocatable;
end subprogram;

```

subprogram ZONE\_6 optional;

\$include(Zone6)

```

module      :FMS:DACS86A0/ROOT.LIB           relocatable;
module      :FMS:DACS86A0/RTHELP286.LIB      relocatable;
module      :FMS:KER286A0/ADAUS.LIB          relocatable;
module      :FMS:ADABIOA1/BINDER286.LIB      relocatable;
module      :FMS:ADARTSA1/BINDER286.LIB      relocatable;
module      :FMS:KER286A0/ADAUS.LIB          relocatable;
module      GATELBA                          relocatable;
end subprogram;

```

subprogram ZONE\_7 optional;

\$include(Zone7)

```

module      :FMS:DACS86A0/ROOT.LIB           relocatable;
module      :FMS:DACS86A0/RTHELP286.LIB      relocatable;
module      :FMS:KER286A0/ADAUS.LIB          relocatable;
module      :FMS:ADABIOA1/BINDER286.LIB      relocatable;
module      :FMS:ADARTSA1/BINDER286.LIB      relocatable;
module      :FMS:KER286A0/ADAUS.LIB          relocatable;
module      GATELBA                          relocatable;
end subprogram;

```

subprogram ZONE\_8 optional;

\$include(Zone8)

```

module      :FMS:DACS86A0/ROOT.LIB           relocatable;
module      :FMS:DACS86A0/RTHELP286.LIB      relocatable;
module      :FMS:KER286A0/ADAUS.LIB          relocatable;
module      :FMS:ADABIOA1/BINDER286.LIB      relocatable;
module      :FMS:ADARTSA1/BINDER286.LIB      relocatable;
module      :FMS:KER286A0/ADAUS.LIB          relocatable;
module      GATELBA                          relocatable;
end subprogram;

```

subprogram ZONE\_9 optional;

\$include(Zone9)

```

module      :FMS:DACS86A0/ROOT.LIB           relocatable;
module      :FMS:DACS86A0/RTHELP286.LIB      relocatable;
module      :FMS:KER286A0/ADAUS.LIB          relocatable;
module      :FMS:ADABIOA1/BINDER286.LIB      relocatable;
module      :FMS:ADARTSA1/BINDER286.LIB      relocatable;

```

```

module      :FMS:KER286A0/ADAUS.LIB          relocatable;
module      GATELBA                          relocatable;
end subprogram;

subprogram ZONE_10 optional;
$include(Zone10)
module      :FMS:DACS86A0/ROOT.LIB           relocatable;
module      :FMS:DACS86A0/RTHELP286.LIB      relocatable;
module      :FMS:KER286A0/ADAUS.LIB          relocatable;
module      :FMS:ADABIOA1/BINDER286.LIB      relocatable;
module      :FMS:ADARTSA1/BINDER286.LIB      relocatable;
module      :FMS:KER286A0/ADAUS.LIB          relocatable;
module      GATELBA                          relocatable;
end subprogram;

subprogram ZONE_11 optional;
$include(Zone11)
module      :FMS:DACS86A0/ROOT.LIB           relocatable;
module      :FMS:DACS86A0/RTHELP286.LIB      relocatable;
module      :FMS:KER286A0/ADAUS.LIB          relocatable;
module      :FMS:ADABIOA1/BINDER286.LIB      relocatable;
module      :FMS:ADARTSA1/BINDER286.LIB      relocatable;
module      :FMS:KER286A0/ADAUS.LIB          relocatable;
module      GATELBA                          relocatable;
end subprogram;

subprogram ZONE_12 optional;
$include(Zone12)
module      :FMS:DACS86A0/ROOT.LIB           relocatable;
module      :FMS:DACS86A0/RTHELP286.LIB      relocatable;
module      :FMS:KER286A0/ADAUS.LIB          relocatable;
module      :FMS:ADABIOA1/BINDER286.LIB      relocatable;
module      :FMS:ADARTSA1/BINDER286.LIB      relocatable;
module      :FMS:KER286A0/ADAUS.LIB          relocatable;
module      GATELBA                          relocatable;
end subprogram;

subprogram ZONE_13 optional;
$include(Zone13)
module      :FMS:DACS86A0/ROOT.LIB           relocatable;
module      :FMS:DACS86A0/RTHELP286.LIB      relocatable;
module      :FMS:KER286A0/ADAUS.LIB          relocatable;
module      :FMS:ADABIOA1/BINDER286.LIB      relocatable;
module      :FMS:ADARTSA1/BINDER286.LIB      relocatable;
module      :FMS:KER286A0/ADAUS.LIB          relocatable;
module      GATELBA
end subprogram;

subprogram ZONE_14 optional;
$include(Zone14)
module      :FMS:DACS86A0/ROOT.LIB           relocatable;
module      :FMS:DACS86A0/RTHELP286.LIB      relocatable;
module      :FMS:KER286A0/ADAUS.LIB          relocatable;
module      :FMS:ADABIOA1/BINDER286.LIB      relocatable;
module      :FMS:ADARTSA1/BINDER286.LIB      relocatable;
module      :FMS:KER286A0/ADAUS.LIB          relocatable;

```

```

module      GATELBA                                relocatable;
end subprogram;

subprogram ZONE_15 optional;
$include(Zone15)
module      :FMS:DACS86A0/ROOT.LIB                  relocatable;
module      :FMS:DACS86A0/RTHELP286.LIB             relocatable;
module      :FMS:KER286A0/ADAUS.LIB                 relocatable;
module      :FMS:ADABIOA1/BINDER286.LIB             relocatable;
module      :FMS:ADARTSA1/BINDER286.LIB             relocatable;
module      :FMS:KER286A0/ADAUS.LIB                 relocatable;
module      GATELBA                                relocatable;
end subprogram;

initial procedure cg_adamainprogram
stacksegment nodal size = 0ff00h;

end program;

hardware configuration;

volume DEFAULT_VOLUME,
presence NODAL_origin FIRST_NODAL_PAGE,
LOCAL to 0 origin FIRST_LOCAL_PAGE,
LOCAL to 1 origin FIRST_LOCAL_PAGE,
LOCAL to 2 origin FIRST_LOCAL_PAGE,
LOCAL to 3 origin FIRST_LOCAL_PAGE,
LOCAL to 4 origin FIRST_LOCAL_PAGE,
LOCAL to 5 origin FIRST_LOCAL_PAGE,
LOCAL to 6 origin FIRST_LOCAL_PAGE,
LOCAL to 7 origin FIRST_LOCAL_PAGE,
LOCAL to 8 origin FIRST_LOCAL_PAGE,
LOCAL to 9 origin FIRST_LOCAL_PAGE,
LOCAL to 10 origin FIRST_LOCAL_PAGE,
LOCAL to 11 origin FIRST_LOCAL_PAGE,
LOCAL to 12 origin FIRST_LOCAL_PAGE,
LOCAL to 13 origin FIRST_LOCAL_PAGE,
LOCAL to 14 origin FIRST_LOCAL_PAGE,
LOCAL to 15 origin FIRST_LOCAL_PAGE;
end configuration;

include ADA286 local to 0;
initial process on 0 priority 2;

include MULTI_ADA286;
$include(SubprogramAssign)
initial process on 0 priority 2;

end system;

```

## 16.2 REFERENCES

[LRM 83]

Reference Manual For the Ada

	Programming Language. ANSI/MIL-STD 1815 A January 1983
[User Guide]	DDC-I Ada 8086/80286 Compiler System User Guide for DACS-80x86 May 31, 1989 DDC-I 5801/RPT/62, issue 12
[DDC3 86] alias [CLU]	DDC-I Ada Compiler System Clusterization Variant of the DACS-80286PM Linker May 31, 1989 DDC-I 5169/RPT/17, issue 1
[MATH]	Mathematics Library for Ada on Mara User Manual rev 1.0
[DUMPS]	Ada Dumps User Manual rev 1.2
[ADAMAP]	ADA MAPPER User Guide
[IDA286]	IDA286 User Guide
[GRP]	GENRP ( MARA286 monograph )
[SCL]	SCL286 ( MARA286 monograph )
[SEL1 87]	MARA-286 An Overview of the Software Factory September 1987
[SEL2 87]	MARA-286 An Overview of the Operating System September 1987
[SEL3 87]	Ada Multi Processor Program Generation November 1989
[SEL4 89]	DDC Ada 80286 Compiler System. Addendum to Appendix F. Supports to MARA Architecture May 5, 1989
[Alenia Software Products]	Basic Software Delivery Document
[MARA286_1 87]	MARA286 Computer Monograph - Volume 1
[MARA286_2 87]	MARA286 Computer Monograph - Volume 2

[MARA286\_3 87]

MARA286 Computer Monograph  
- Volume 3

[MARA286\_4 87]

MARA286 Computer Monograph  
- Volume 4